Enforcing Abstract Immutability

by

Jonathan Eyolfson

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Ana Milanova Associate Professor Rensselaer Polytechnic Institute
Supervisor	Patrick Lam Associate Professor University of Waterloo
Internal Member	Lin Tan Associate Professor University of Waterloo
Internal Member	Werner Dietl Assistant Professor University of Waterloo
Internal-external Member	Gregor Richards Assistant Professor University of Waterloo

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Researchers have recently proposed a number of systems for expressing, verifying, and inferring immutability declarations. These systems are often rigid, and do not support "abstract immutability". An abstractly immutable object is an object *o* which is immutable from the point of view of any external methods. The C++ programming language is not rigid—it allows developers to express intent by adding immutability declarations to methods. Abstract immutability allows for performance improvements such as caching, even in the presence of writes to object fields. This dissertation presents a system to enforce abstract immutability.

First, we explore abstract immutability in real-world systems. We found that developers often incorrectly use abstract immutability, perhaps because no programming language helps developers correctly implement abstract immutability. We believe that this omission leads to incorrect usages. Specifically, we wrote a dynamic analysis that reports any writes through immutability declarations. To our knowledge, this work was the first to explore how objects implement abstract immutability (or fail to implement it). Our novel study found three uses of abstract immutability: caching, delayed initialization, and unit testing. Unit testing was a surprising application of abstract immutability, and we believe that the ability to modify state is needed for proper unit testing.

Next, we explore developers' revealed needs for immutability in the source code. We found that the majority of classes contain a mix of immutable and mutable methods, with a majority of the overall methods being immutable. Immutability systems with only immutable or all-mutating classes are insufficient: developers need immutability declarations at method granularity. Our study then combined developer immutability declarations with results from a static analysis to estimate the true number of immutable methods. The static analysis checked that no transitive writes to a receiver object occurred. Our results indicated the need for a sophisticated analysis to check that these apparently abstractly immutable methods were indeed abstractly immutable.

Finally, we created a novel static analysis which checks that developers follow abstract immutability. Specifically, we define abstract immutability to mean that a class's set of immutable methods is collectively free of writes to exposed fields. Our analysis found incorrect usages of abstract immutability, such as incorrect caching. This analysis is particularly valuable in the context of code evolution, whereby subsequent programmers may make changes that break previously-correct cache implementations, for instance. Our work allows developers to trust that their code is abstractly immutable.

Acknowledgements

First, I need to thank my parents. Especially my dad, who has supported me to follow my passion no matter what. Whether I'm at my highest highs or lowest lows, he has guided me through them all. Thank you Grandma Cook, you've always ensured I was taken care of and I couldn't have done it without you. I would also like to thank Tracy for all her love and support, and putting up with horrible puns.

I would be remiss if I didn't thank my teachers at Frontenac High School: Mr. Kotecha, Mr. Wills, Mr. Lee, and Mr. Grew. I would not have found my passion if it wasn't for them opening my eyes, and setting an excellent example. They encouraged the pursuit of knowledge, and curiosity I didn't know I had.

I owe a deep gratitude to my supervisor Patrick Lam, who has been guiding me ever since we first met at Curry Original in Kingston. Without his knowledge and expertise I would not have completed my masters, much less a doctorate. He has been patient and understanding through my graduate studies, and always allowed me to explore problems that interested me.

Thank you to my committee members. I appreciate the time you've taken out of your busy schedules to add your valuable insight to this work.

Finally, I would like to thank friends that I've been able to bounce ideas off of during my research. There are too many to name, you know who you are, thank you. For the purpose of this thesis in particular: Jean-Christophe Petkovich, you've been invaluable, as both a set of open ears and a friend dating back from grade 3.

Dedication

In memory of Sisters Kathryn and Mary Joan LaFleur.

Table of Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

Mutation in software is challenging to reason about. Following and understanding mutations is required for debugging many programs. Restricting mutation of a variable `x` therefore enhances the ability of maintainers to reason about `x`. Making `x` immutable reduces developers' cognitive load, as the developer does not have to worry about unexpected changes to `x`.

However, even objects marked immutable may change internally. The canonical example of internal mutation is caching. Performant code relies on these types of writes. Unfortunately, such writes make it especially challenging to reason about immutability. Aside from caching, other instances of immutability in the presence of writes is unclear.

The central concept of this dissertation is *abstract immutability*. Caching is an example of abstract immutability. A caching implementation requires a write which is invisible to the caller. A method in a class may internally mutate, and yet be callable on an immutable receiver object. The caller would observe no difference between a bitwise immutable object and an abstractly immutable object, aside from the performance benefit.

C++ allows developers to restrict mutation by including a `const` type qualifier in a variable's type. A `const`-labelled variable cannot be written to and should not change. Properly using `const` gives 1) library users static guarantees about how the code behaves and 2) compilers opportunity to more aggressively optimize the code. C++ experts recommend using `const` whenever possible [52]. In addition, developers are told to strive for `const` correctness [24].

For instance, if a method `bar` acts on an object `x` with a `const` qualifier, the caller can expect that `bar` will not modify the `x`. If the author of `bar` attempts to modify the object, the compiler produces an error. When developers use `const` correctly, their code is understood at a glance with respect to mutation, and the type checker will not allow invalid code to compile.

However, functions such as `bar` may modify the object though one of many escape hatches in C++. When writing, for instance, method that implements caching, developers require these escape hatches to perform any writes. These writes need to be allowed. However, although the writes are allowed, the compiler does not check that the writes are valid. The method author could later add an unintended write, breaking immutability, without any indication from the compiler. This would produce unexpected results if a reader expected the immutable object not to change.

Listing 1.1: Developers would expect this program to return success.

```
#include <unordered_set>
#include "Foo.hpp"

int main(int argc, char *argv[])
{
    const Foo f(7);
    std::unordered_set<Foo> s;
    s.insert(f); // copies f into the s container
    f.bar(); // bar is a const member function
    if (s.count(f) > 0) return EXIT_SUCCESS;
    else return EXIT_FAILURE;
}
```

1.1 Motivation

A `const` declaration should mean that the declared `const` object should never observably change. It is especially important that libraries are `const` correct so library users may use them without any unexpected behaviour. For instance, if a library function takes a `const` reference to an object, the library user can expect the library not to call non-`const` member functions on the object. Consider the code in Listing 1.1 which uses a class `Foo` declared in a library.

This simple program creates a `const Foo` called `f`. Since `f` is declared as `const` it should not change for the duration of the program, or at least appear not to change from the point of view of the library user. The program then adds a copy of `f` to set `s`. Afterwards, there is a call to a `const` member function on `f`. Since `f` is unmodified still, it should still be present in the set and the program should return success. While these assumptions should be correct, the program could still return failure. Consider the following, valid C++, definition of `Foo` in Listing 1.2.

We see that class `Foo` is a wrapper around an `int` variable. Its `int` field is declared as `mutable`, which allows that field to change even in `const` member functions. The `bar` function (or any other function) may now modify the field without any checks for `const`, changing the meaning of the object. `Foo` successfully compiles, but what we expect from the original code no longer holds, causing the program in Listing 1.1 to return failure. Reasonable people would not consider this `Foo` to be logically `const`. Specifically, after the call to `f.bar()` in the example, `f`'s wrapped `int` value is 42. The copy inserted into `s` still has an `int` value of 7. This causes the lookup to fail. Our novel static analysis can analyze such code and report an error.

Another problem with `const` is that a library with `const` incorrectly omitted by its authors prevents the library's users from including `const` in their own code. Consider the definition of `Date` inspired from Stroustrup's book [75] in Listing 1.3. This class uses caching internally: it guards the cached value with the `cache_valid` boolean, and writes to `cache_rep`. However, the caching is an implementation detail invisible to users of

Listing 1.2: A `Foo` implementation that causes unexpected results.

```
class Foo;
template<> struct std::hash<Foo> {
    size_t operator()(const Foo& f) const;
};
class Foo
{
public:
    Foo(int i) : d(i) {}
    void bar() const { d = 42; }
private:
    mutable int d;
    friend bool operator==(const Foo& f1, const Foo& f2);
    friend size_t std::hash<Foo>::operator()(const Foo& f) const;
};
inline bool operator==(const Foo& f1, const Foo& f2)
{ return f1.d == f2.d; }
inline size_t std::hash<Foo>::operator()(const Foo& f) const
{ return std::hash<int>()(f.d); }
```

Listing 1.3: A `Date` class that incorrectly omits `const` on `string_rep`.

```
#include <cstdlib>
#include <string>

class Date {
    bool cache_valid;
    std::string cache_rep;
    void compute_cache_value()
    {
        cache_rep = std::string(1, std::rand() % 26 + 65);
    }
public:
    std::string string_rep() // ought to be const
    {
        if (cache_valid == false) {
            compute_cache_value();
            cache_valid = true;
        }
        return cache_rep;
    }
};
```

this class. Users would correctly assume that the string representation of a date which remains unmodified is also unmodified, even though a write to a field may occur in the implementation. A `const` correct implementation of `Date` would change `string_rep` to a `const` member function. Therefore this `Date` class needs to use either `const_cast` or `mutable` when writing to `cache_rep` to subvert the restrictive `const`. As seen in Listing 1.2, `mutable` can easily be misused (similarly, so can `const_cast`), especially with evolving code adding more functions.

1.2 Case Study: Clang

On November 4th, 2013 we manually inspected a portion of Clang's code looking for changes which would improve `const` correctness. In the `CFGBlock` class we found two versions of a `getTerminatorCondition` method: a non-`const` version returning a non-`const` pointer and a `const` version returning a `const` pointer. The `const` version uses the non-`const` code by casting away `const` on `this`.

We believe it is clearer to write a `const` (more restrictive) version of the function and have the non-`const` version reuse the `const` version, while using `const_cast` on its return value. In this specific case we observed that all internal client code did not attempt to modify the returned `Stmt` indicating the (unused) non-`const` version could be removed.

We submitted the patch in Listing 1.4 with the intention of receiving valuable insight from developers of a large widely used body of code regarding `const`. We modified the `CFG` class to remove a non-`const` definition of a member function and replace it with a `const` one. This class is used by external analysis tools, as well as Clang's own internal analysis. While we did not have any external analysis tools, our more restrictive `const` correct version did not create any errors when compiling Clang itself. Refer to Appendix A for full discussion on the mailing list.

The feedback we received indicates that although developers use `const`, its usage is not consistent throughout the code. Our suggested changes disallows users from modifying the AST in the `CFG` class. However the developers have not solidified how their library should be used. Our experience indicates that even large well-constructed programs have many improvements to make in regards to `const` correctness.

1.3 Exploratory Study

Our initial hypothesis is that `const` is a widely used and important factor in library design. We conducted an exploratory study on LLVM/Clang to observe `const`'s usage in a real-world application. We counted the number of non-`const`/`const` methods and non-`mutable`/`mutable` fields to compare how often each keyword occurs. We tracked these numbers across multiple LLVM/Clang versions to observe any trends. In addition, we separated "include" code (which is meant for clients of LLVM/Clang) and "application" code (which we observed typically is only used in one or two internal libraries). Specifically, "include" code is used by any external code linked to the LLVM library, while "application" code is private and is only used in the internal library code. Table 1.1 shows our results.

Listing 1.4: Our patch improving Clang's `CFG` class `const` correctness.

```
diff --git a/include/clang/Analysis/CFG.h b/include/clang/Analysis/CFG.h
index d42b19e..9236bce 100644
--- a/include/clang/Analysis/CFG.h
+++ b/include/clang/Analysis/CFG.h
@@ -511,11 +511,11 @@ public:
   CFGTerminator getTerminator() { return Terminator; }
   const CFGTerminator getTerminator() const { return Terminator; }

-  Stmt *getTerminatorCondition();
+  const Stmt *getTerminatorCondition() const;

-  const Stmt *getTerminatorCondition() const {
-    return const_cast<CFGBlock*>(this)->getTerminatorCondition();
-  }
+  // const Stmt *getTerminatorCondition() const {
+  //   return const_cast<CFGBlock*>(this)->getTerminatorCondition();
+  // }

   const Stmt *getLoopTarget() const { return LoopTarget; }

diff --git a/lib/Analysis/CFG.cpp b/lib/Analysis/CFG.cpp
index 14b8fd4..6f50f49 100644
--- a/lib/Analysis/CFG.cpp
+++ b/lib/Analysis/CFG.cpp
@@ -3950,12 +3950,12 @@ void CFGBlock::printTerminator(raw_ostream &OS,
   TPrinter.Visit(const_cast<Stmt*>(getTerminator().getStmt()));
 }

-Stmt *CFGBlock::getTerminatorCondition() {
-  Stmt *Terminator = this->Terminator;
+const Stmt *CFGBlock::getTerminatorCondition() const {
+  const Stmt *Terminator = this->Terminator;
   if (!Terminator)
     return NULL;

-  Expr *E = NULL;
+  const Expr *E = NULL;

   switch (Terminator->getStmtClass()) {
     default:
```

Table 1.1: Number of non-`const`/`const` methods and non-`mutable`/`mutable` fields between LLVM/Clang versions.

LLVM/Clang Version	Methods (non-`const`)	Methods (`const`)	Fields (non-`mutable`)	Fields (`mutable`)
2.9	21,293	15,587	13,331	269
(include)	10,648	10,599	7,594	143
(application)	10,645	4,988	5,737	126
3.0	23,660	17,441	15,185	319
(include)	11,760	12,292	8,711	160
(application)	11,900	5,149	6,474	159
3.1	25,579	19,131	16,626	333
(include)	12,604	13,544	9,553	173
(application)	12,975	5,587	7,073	160
3.2	28,413	20,834	19,150	365
(include)	13,381	14,690	11,124	200
(application)	15,032	6,144	8,026	165
3.3.1	30,361	22,583	19,736	389
(include)	14,060	15,441	10,781	215
(application)	16,301	7,142	8,955	174
3.4.2	32,347	22,977	21,080	390
(include)	14,410	15,069	11,192	211
(application)	17,937	7,908	9,888	179

We found that `const` accounts for over 40% of the methods in LLVM/Clang across all versions. In general, for code intended to be used by clients, over 50% of the methods include `const`. For fields, `mutable` is used far less, and split evenly between include and application code. This initial exploration sparked the core idea for this dissertation, to investigate immutability.

1.4 Outline

The thesis of this dissertation is that abstract immutability is important to developers and checkable via static analysis. First, to establish a common understanding of immutability, we provide background on immutability including generic terminology and how it applies to C++ specifically in Chapter 2. Our next goal is to answer whether developers use abstract immutability. What do they use it for? We developed a dynamic analysis to address this question in Chapter 3. In a broader sense, we would also like to know whether developers care about immutability. For instance, is it important enough to be worth the effort to use? We investigate several codebases to determine the amount of immutable code in Chapter 4. Finally, we check abstract immutability with a novel static analysis in Chapter 5.

Chapter 2

Background

Many definitions for immutability exist. Its definition can vary from developer to developer, and from programming language to programming language. This chapter breaks down the general concept of immutability into different subconcepts: 1) transitivity, 2) abstractness, and 3) object/reference. We explain how these general concepts relate to what is currently available in programming languages. Our work uses these immutability concepts and leverages both dynamic and static analysis to detect and enforce immutability properties.

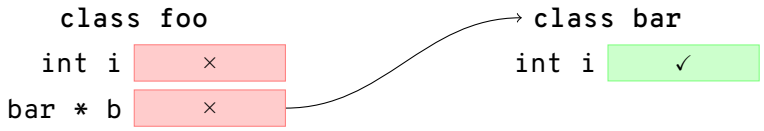
2.1 Immutability Terminology

We explore immutability applied to objects and primitive types, and in particular, we describe the three most important aspects that are key to understanding our work. Note that our generic terms differ from those of immutability in Java [14]. We use the generic term pointer to match that of a C pointer; we would refer to a Java reference as a pointer to an object.

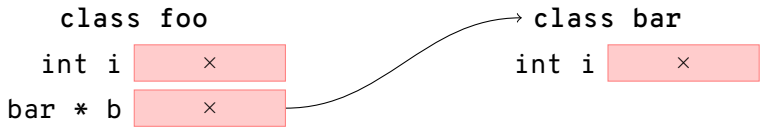
2.1.1 Transitive and Non-Transitive

For a class, transitivity describes how immutability applies to its fields. For instance, consider a class with two fields: a pointer to an object and a primitive type. A class that has non-transitive (and transitive) immutability protects its fields from re-assignment; changes to primitive values or pointer manipulations are not permitted. However, a class with non-transitive immutability allows writes to values reachable through a pointer. Deep (or transitive) immutability means that the class prevents modifications to any state reachable through its fields. C++ supports non-transitive immutability.

Figure 2.1 visually illustrates transitivity. Figure 2.1a shows non-transitive, where fields of the `foo` object may not be modified (shown in red with an ×). The `bar` object pointed to by the field `b` may be modified (shown in green with a ✓). Figure 2.1b shows the transitive case: neither the field nor anything reachable through the field may be modified. For `foo` in this example, given a reference to a `foo`, it is not possible to modify the linked `bar` through its `b` field.



(a) Non-transitive immutability.



(b) Transitive immutability.

Figure 2.1: With non-transitive immutability, fields may be dereferenced and written to. Transitive immutability disallows all writes (fields shown by × cannot be written to while fields shown by ✓ may).

2.1.2 Concrete and Abstract

Concrete and abstract immutability determine whether writes are allowed. Concrete immutability is what first comes to mind as the meaning of immutability for most developers; it disallows any writes that immutability applies to. However, consider an object with a cache. Caching requires writes to implement. Yet the object could still be immutable if its observable state does not change. Abstract immutability allows writes that don't change the behaviour of a class. The notion of abstract immutability exists in the literature. Languages provide some support for concrete immutability. Our work is the first to provide an operational definition for abstract immutability.

The literature [61] contains examples of valid abstract immutability. For instance, consider a class representing a tree data structure. Some operations could reorganize a tree by rebalancing it, violating concrete immutability since writes occur. However, these operations do not change the contents of the tree. Prior to this work, it was the developer's sole reposibility to ensure their writes preserved abstract immutability. We investigate real world cases where programs violate concrete immutability in Chapter 3.

2.1.3 Object and Reference Immutability

Object immutability relates to an instance of class over its lifetime. If a class has class immutability, then every instance of that class is immutable. Some languages, such as Ruby, allows mutable objects to be frozen (see further dicussion in Chapter 6.3.2). When an object is frozen, it becomes an immutable object for its remaining lifetime. The definition of immutability can vary, as outlined in the previous two subsections.

For reference immutability, developers can declare a pointer to an immutable type. Typically the type system allows developers to add an "immutable" type qualifier. A pointer to an immutable type can only call a set of methods which are marked as being immutable. A pointer to a mutable type may call methods marked as immutable along with any unmarked (hence potentially mutating) methods. C++ supports reference immutability. Note that the set of callable methods for a pointer to a non-immutable type need not be a superset of methods available to a pointer to an immmutable type: the set of methods may be disjoint.

If there is a pointer to an immutable type, reference immutability indicates that the object will not change through that pointer. The difference between reference immutability and object immutability is that there may be pointers to a non-immutable type that alias pointers to an immutable type. In this case, an object may visibly change through a pointer to an immutable type due to a change through a non-immutable pointer. Our work provides an analysis for abstract immutability which provides checks for a set of methods marked as immutable.

2.1.4 Relation to Purity

The term purity indicates that a function has no side effects. Informally, the function does not modify any state within the program. In the strictest sense, a "pure" function may not modify any variables. Less strict versions of purity may modify fresh variables. Purity only applies to functions. It does not take encapsulation into account and does not have an analogy to abstract immutability.

Observational Purity

Observational purity is less strict than purity. Intuitively, it considers methods pure if any changes they make cannot be observed in the program. It is an extension of "weak" purity. Weak purity prevents any location in the pre-state from being modified. A location is modified if it's available in both the pre and post-state and has a different value in the post-state. This definition of weak purity allows a value to change and change back within a method. The definition also allows newly allocated objects within a method to be modified.

Observational purity is when a method is "weakly" pure from the point of view of the caller, but may modify private data [6, 56]. For a method to be observationally pure, it must not modify any locations in the pre and post-state that are available to the caller. This allows writes to private fields. However, these writes to private fields aren't checked. There is only a rule that private fields cannot escape the method.

"Benign" side effects. These are side effects that do modify values, but do not change the behaviour of the program. Such effects should be allowed by observational purity. For instance, caching modifies values, but properly encapsulated caching through the immutable interface is invisible to clients of the object [12]. Our work to check abstract immutability is similar to checking "benign" side effects for objects.

2.2 Immutability in C++

In C++ developers specify immutability using the `const` type qualifier. At its core, `const` does not have a specific meaning: while type rules govern its use, developers are free to use it to specify any kind of immutability. The type system provides no guarantees for `const` qualified types, and cannot be leveraged by compilers, e.g. for optimizations.

In this section we explain how `const` behaves in C++. To allow developers to write abstract immutability, one of the facilities C++ provides is the `mutable` keyword. We explain this keyword and other ways to implement abstract immutability in C++. We relate commonly used C++ terms to generic immutability terms and finally show that C++ developers do use `const` often in practice.

2.2.1 `const` keyword

Recall that the C++ `const` keyword allows programmers to declare, in some sense, that a value should not change. In this section, we explain the specific guarantees that C++ provides.

The keyword `const` has two usages in C++: it can be either 1) a type qualifier (e.g. `const int x`), or 2) a method specifier (e.g. `void foo(int y) const`). As a type qualifier, `const` always applies to the type to its left. If there is no type to the left, it applies to the right.

Meaning of `const` on C++ primitive and pointer types. The meaning of `const` in C++ is an extension of its meaning in C. We start by describing the common meaning of `const` across C and C++, applying to primitive and pointer types. In these languages, the `const`-ness of a memory location depends on the qualifiers of the variable through which the location is accessed.

Developers may `const`-qualify primitive types such as `int`, resulting in immutable object types like `const int`. When variable v has primitive `const`-qualified type, the C++ type system prevents developers from assigning to v after its definition; i.e. it prevents writes to v.

For pointer types, such as `int *`, developers may `const`-qualify both the pointee and pointer type. The `const` qualifier applies to the type directly to the left of it; if there is nothing to the left, then `const` applies to the right. If a variable has type `int *const`, developers may not change the address value of the variable (where it points to), but they may dereference the location and change the value it points to. A different type is `int const *` (also known as `const int *`), which allows the address value to change, but not the value pointed-to by the variable. The `const` qualifier may also apply to both the pointer and pointee types (`int const * const`), which prevents writes to both the address value and the value pointed-to. If all pointers to a value are to an immutable type, then the value pointed-to is immutable. C++ references can be thought of as `const`-qualified pointers; a developer may not write to the reference address value. However, in contrast with pointers, developers cannot cast away reference address value `const`-ness and re-assign the address value; this property is enforced by the language.

Meaning of `const` on C++ object types. We continue by exploring the meaning of `const` in C++-specific contexts. When a C++ object type is `const`-qualified, the developer may only call methods declared with a `const` qualifier.

`const`-qualifying a member function has two effects. First, `const`-qualifying a member function allows it to be called on a `const`-qualified receiver object. Furthermore, inside the function, the type qualifiers of the receiver object's fields are treated as `const`.

Listing 2.1: The `const` qualifier may apply to C++ member functions.

```
class Pointish {
private:
  int x;
  int * y;
public:
  int getX() const { return x; }
  void setX(int val) { x = val; }
  void setY(int val) { *y = val; }
};
```

not OK if `setX()` were `const`

transitive write
OK if `setY()` were `const`

Conceptually, each C++ class provides two interfaces: the `const`-qualified interface and the non-`const` qualified interface. A `const`-qualified reference is meant to be a read-only reference, although C++ enforces no guarantees. One of our goals is to evaluate whether read-only guarantees hold in practice. When an object has non-`const` type, then the developer may call all methods on that object[1]. On the other hand, when an object has `const` type, then the developer may only call methods on that object that are `const`-qualified.

Consider class `Pointish`, defined in Listing 2.1. As written, the developer could call all methods on a non-`const`-qualified object of type `Pointish`. On the other hand, the developer may only call the `getX()` method on a `const`-qualified `Pointish` object.

The second effect of `const`-qualifying a function changes the type qualifiers of fields inside the function. In our example, field `x` becomes `int const` within `const`-qualified member functions. The compiler successfully compiles `getX()`, since there are no writes to `x` or `y`. But, if `setX()` was `const`-qualified, the compiler would refuse to compile the code, since the type of `x` would be treated as `const int` and `setX()` contains a write to that variable.

In C++, without using `const` escape hatches, developers may re-assign fields in non-`const` qualified methods and may not re-assign fields in `const`-qualified methods. In all methods, developers are permitted to mutate state outside of re-assignment (through references or pointers). This type of immutability is *non-transitive concrete immutability*.

A C++ `const`-qualified stack/global object would be considered a shallow *immutable object*. That is, without escape hatches, developers cannot create non-`const` references (including through pointers) to such a `const`-qualified object. However, as we discuss next, developers may indeed remove the `const` qualifier on references to the `const`-qualified object. Therefore, C++ does not strongly enforce the concept of an immutable object.

2.2.2 `mutable` keyword

The keyword `mutable` is a storage specifier used to specify class/struct fields which may change within a `const` qualified method. That is, any field with a `mutable` specifier does

[1]There is a small exception: on a non-`const` object, the developer cannot call `const`-qualified methods that are hidden due to overloading by a non-`const`-qualified method of the same signature.

Table 2.1: C++ only prevents non-transitive field writes without `mutable`, it allows all other writes.

Field declaration	Statement in `const` method	Permitted
`int * i`	`i = ...`	Disallowed
`int * i`	`*i = ...`	Allowed
`mutable int * i`	`i = ...`	Allowed
`mutable int * i`	`*i = ...`	Allowed

not get a `const` qualifier implicitly applied to its type within the context of a `const` qualified method.

A C++ `mutable` field is never interpreted with `const`, even in `const` methods. Conversely, fields including `const` always have `const` applied to them, even in non-`const` methods. Unlike `const`, `mutable` is a *storage specifier* and only applies to the memory used by the field itself. This means if a `mutable` field is a pointer, the pointer value can change. Note that a pointer field can always change the value being pointed to. All possible usages of `mutable` with `i` are shown in Table 2.1.

2.2.3 Bitwise `const`

In C++ the term bitwise `const` refers to non-transitive concrete immutability. Bitwise `const` means that, through an object's `const` interface, the values of its field do not change. In other words, the bits used to store the object do not change. In this case, a `const` object's methods may modify other objects, as long as it does not store the change on its own fields. For instance, it may modify a second object through its pointer field, and as long as the pointer value of its own field does not change, it is still considered bitwise `const`.

2.2.4 Logically `const`

Logically `const` is, in practice, the goal of `const` qualified methods [52], which is why `mutable` exists in C++. In general terms, if a class is logically `const`, then that the class is transitively and abstractly immutable.

2.2.5 Additional Loopholes

C++ [72] is almost source-level compatible with C. However, this source-level compatibility allows several mechanisms for subverting the type system and discarding `const` type qualifiers. The mechanisms are as follows: unchecked casts, unions and varargs.

Pointers. Listing 2.2 shows one way to subvert the example code from Listing 1.3 in existing C++. Recall that Listing 1.3 has a logically `const` member function `string_rep` which ought to, but does not, include `const` in its declaration.

Listing 2.2: A `Date` class, using pointers, that allows `const` on `string_rep` method without using `mutable` or `const_cast`. `Date` follows abstract and transivite immutability.

```
#include <cstdlib>
#include <string>

struct cache {
    bool valid; // previously cache_valid field in Date
    std::string rep; // previously cache_rep field in Date
};

class Date {
    cache * c; // replaces cache_valid and cache_rep fields
    // c's fields may be changed via dereferences in
    // const member functions
    void compute_cache_value() const
    {
        c->rep = std::string(1, std::rand() % 26 + 65);
    }
public:
    // c's fields may be changed via dereferences in
    // const member functions
    std::string string_rep() const
    {
        if (c->valid == false) {
            compute_cache_value();
            c->valid = true;
        }
        return c->rep;
    }
};
```

Listing 2.3: A `Date` class inappropriately, but not incorrectly, allowing `const` on the `string_rep` method through `const_cast`.

```
#include <cstdlib>
#include <string>

class Date {
    bool cache_valid;
    std::string cache_rep;
    void compute_cache_value()
    {
        cache_rep = std::string(1, std::rand() % 26 + 65);
    }
public:
    std::string string_rep() const
    {
        if (cache_valid == false) {
            // remove the const qualifier
            Date * t = const_cast<Date *>(this);
            t->compute_cache_value();
            t->cache_valid = true;
        }
        return cache_rep;
    }
};
```

There are minor differences between Listing 1.3 and Listing 2.2. Listing 2.2 1) declares a `struct cache` with `rep` and `valid` fields, 2) replaces fields with a single field (a pointer to `struct cache`), and 3) replaces all field reads/writes by referencing the pointer field.

The current type system allows `string_rep` to include `const` by adding one layer of indirection (the pointer to the `struct cache`). We want to make the type system consistent. If Listing 1.3's `string_rep` can not include `const` without explicitly subverting the type system, Listing 2.2 should not be able to include `const` on its `string_rep` either. However, Listing 2.2 type checks because `const` member functions only apply `const` to the pointer value, not what it points to.

There are two writes allowed by the type system in `Date::string_rep`. Our analysis will subsequently verify these writes to `c->rep` and `c->valid`.

Casts. C casts are unchecked, and may remove `const` qualifiers. C++ introduces 4 types of casts: `static_cast`, `dynamic_cast`, `reinterpret_cast`, and `const_cast`. Only one type of cast, `reinterpret_cast`, is unchecked (equivalent to C style casts).

Listing 2.3 shows a way to subvert the type system using `const_cast`. This example casts away the implicit `const` of `this` for `const` methods and writes to fields.

Some developers have a misconception that downcasts are not dynamically checked for

Listing 2.4: C style downcasts are not checked for `const`, C++ style dynamic casts are.

```
class A {
public:
    A() {}
};
class B : public A {
public:
    B() {}
};
int main(int argc, char * * argv)
{
    const B b;
    A * pa;
    pa = (A *) &b; // allowed
    pa = dynamic_cast<A *>(&b); // disallowed
// compile error: dynamic_cast from 'const B *' to 'A *' casts away
//                qualifiers
    return 0;
}
```

`const`. C++ style `dynamic_cast`s, however, are statically checked for `const` and produce an error at compile-time. This is possible since C++ encodes a type's qualifiers in its Run-time Type Information (RTTI). See Listing 2.4 for an example.

Unions. Unions allow mutually exclusive variables to share the same storage space. Each variable name within a union can have any type. A trivial way to subvert `const` in this case is if a union has two variables with the following types: `const` and non-`const`. In this case a developer can modify the `const` variable name through its non-`const` name.

Varargs. C style varargs are unchecked variable-length procedure arguments. They have major disadvantages: 1) argument types are unchecked by the type checker and 2) they only support Plain Old Data (POD) types. A POD type only contains fields (it may not have a constructor/destructor or member functions). C++11 initializer lists solve both of these issues: argument types are checked, and classes are supported. See Listing 2.5 for an example of C style varargs allowing unchecked casts while C++11 initializer lists do not.

2.3 Immutability in Java

Java has limited support for immutability by default. Immutability present in Java is object immutability, where all instances of a class are immutable. A typical example of this is the `String` class. Sometimes there are two versions of classes, one immutable and the other allowing mutation (for example, `StringBuilder`). These classes may or may not

Listing 2.5: C style varargs are not checked for `const`, C++11 style are.

```
#include <cstdarg>
#include <initializer_list>

class A {
public:
    A() {}
    void non_const() {}
};

void c_varargs(int n, ...)
{
    va_list list;
    va_start(list, n);
    for (int i = 0; i < n; ++i) {
        A * a = va_arg(list, A *);
        a->non_const();
    }
    va_end(list);
}

void cpp_initializer_list(std::initializer_list<A *> list)
{
    for (auto a : list) {
        a->non_const();
    }
}

int main(int argc, char * * argv)
{
    const A a1;
    const A a2;
    c_varargs(2, &a1, &a2); // allowed
    cpp_initializer_list({&a1, &a2}); // disallowed
// compile error: no matching function for call to 'cpp_initializer_list'
// note: candidate function not viable: 1st argument ('const A *')
//       would lose const qualifier
}
```

be related through the inheritance hierarchy. Java also has the `final` keyword (discussed below), and extensions such as Javari which are more similar to C++.

2.3.1 `final` Keyword

The `final` keyword on variables in Java disallows reassignment. This behaves exactly the same as C++ `const` when applied to a primitive type. Using `final` on every field of an object gives the same effect as a `const` C++ method without `mutable`. In our terms this is non-transitive concrete immutability.

2.3.2 Javari

Javari [83, 82] has a `readonly` keyword which has similar goals to C++ `const` (except for fields it applies transitively). Class methods may be qualified `readonly`, allowing `readonly` objects to call them. Like C++, Javari also includes a `mutable` keyword to support abstract immutability. Unlike C++, Javari does not include additional loopholes for casting and pointers.

2.4 Static Analysis

Our approach will primarily use static analysis to check that code follows our definition of abstract immutability. Static analysis approximates execution traces of a program [54]. The two major techniques we use in our static analysis are dataflow and pointer analysis.

2.4.1 Dataflow Analysis

Dataflow analysis is a form of static analysis. A user of dataflow analysis proceeds as follows [43]. First, define a property (such as number signs) and approximate its values using a lattice. Next, using the Control Flow Graph (CFG), estimate the value of the property at all program points. Read off selective points of interest.

This technique is thoroughly researched, widely used, and has frameworks for implementing the analysis (such as Clang). Specifically, we present our analysis in the style of Nielson, Nielson, and Hankin [58].

We chose dataflow analysis in particular because the properties we wish to express should hold over every execution of the program and are closely related to the static type system. Note that dataflow analysis is conservative since it must make imprecise assumptions.

The first design decision when formulating a dataflow analysis is to pick a lattice. Essentially, a lattice approximates some set of program values for an analysis. A standard example is a powerset lattice. A powerset lattice includes all combinations of possible values drawn from a fixed set. This means that at a particular program point there may be no values or every value may exist (or any combination of values).

There are several ways to traverse the CFG for the analysis. The technique we plan to use is a forward analysis. Forward analysis starts at the entry node of the CFG and moves

towards the exit nodes (as a normal execution would). This computes facts about the past. Our analysis computes sets of program values using *may* information with *kill* and *gen* transfer functions. A transfer function defines the values that are removed (in the case of *kill*) or added (in the case of *gen*). The analysis computes least sets so we can ensure that if a value is available in one execution path, it may be available at the exit node.

A worklist algorithm calculates the values at all program points. The worklist contains a list of program points that need to be calculated using the transfer functions. Each iteration refines the values at that program point and adds other program points that program point depends on to the worklist. The algorithm terminates when the worklist is empty, at which point the dataflow analysis has reached the fixed point.

Our analysis will also be path-sensitive. This means we use the information embedded in conditionals when analyzing CFG blocks for both branches. Clang provides symbolic conditional values as part of its static analysis framework.

Our analysis is context sensitive. A context sensitive analysis uses information at a call site when analyzing a function. For `const` methods, calls within that function's body take context information into account in our analysis. Enabling context-sensitivity requires our analysis to be interprocedural.

2.4.2 Pointer Analysis

Because C++ programs use pointers, we need alias information to properly analyze real programs. In particular, since a program may take an address of a field, our analysis needs to determine if a pointer aliases a field or not.

Pointer analysis answers whether two pointers must, may, or may not alias each other. The common way to do pointer analysis is to first label all the static allocation sites within the program and then to apply one of several analyses [34], e.g.: Andersen's Algorithm (AA) or Steensgaard's Algorithm (SA). AA uses a flow and context insensitive dataflow analysis and computes pointer constraints for the entire program. There are variants of AA which ensure fields of the same class do not alias [32]. SA also uses a flow and context insensitive dataflow analysis. However, SA uses a "union-find" data structure instead of constraints. This data structure makes imprecise assumptions by merging pointer values. SA has less overhead than AA but is also less precise. Clang/LLVM implement SA [79]. Another variant is object sensitive analysis [53]. Object sensitive analysis allows the analysis designer to distinguish different object types at static allocation sites. Due to our abstraction we integrate pointer analysis with our state.

Chapter 3

Dynamic Observations of Writes-Through-Immutability

Immutability, at its core, specifies that a variable does not change. So, if variable *o* carries immutability declarations, one might expect that no writes occur to *o*. In this chapter we explore writes-through-immutability declarations specified by developers. Specifically, we use C++'s `const` qualifier as it is a widely used immutability declaration. Throughout this chapter we refer to any writes-through-immutability specifically as writes-through-`const`.

We distinguish between two uses of C++'s `const` qualifier: `const`-qualified global or stack objects, whose data may never change (i.e. immutable objects), and `const`-qualified references/pointers to objects (i.e. read-only references [61]). (For now, assume that there are no casts that remove `const` qualifiers and no `mutable` storage class specifiers; these language features violate bitwise immutability.) An *immutable object*'s fields may never change, and mutable references to that object may never be created. A *read-only reference*, on the other hand, only guarantees immutability for accesses through qualified references. An object with a read-only reference to it may still be mutated through other, mutable, references. C++ enforces a shallow immutability guarantee (also known as bitwise `const`) for writes through the read-only reference: while it is illegal to reassign the fields of such an object, any referents of fields may change. We expanded on this in Chapter 2.

C++'s type system admits several workarounds to `const`'s supposed immutability guarantees. Much research defines type qualifiers similar to C++ `const`, but with stronger guarantees. These type systems do not have any holes in the type system, such as unsafe casts. Furthermore, they not only ensure that the field values do not change, but also ensure that objects referenced through fields also do not change (i.e. deep, or transitive, immutability; again, see Chapter 2 for more details).

On the other hand, our industrial contacts have indicated that, in their codebases, `const` has been used to deter new developers from modifying certain variables. Such variables may be modified by an experienced developer ready to assume the consequences [21]. In such cases, `const` serves an advisory role, but does not provide any guarantees.

Our goal in this chapter is to explore the space of possible meanings for immutability declarations in C++ and to examine what guarantees developers appear to be expecting in practice. This allows us to observe any instances of abstract immutability. We developed a tool, ConstSanitizer, that instruments programs to identify source locations that modify

`const`-qualified objects, using more restrictive semantics than guaranteed by C++. ConstSanitizer monitors writes-through-`const`, i.e. writes performed on `const`-qualified objects or references, either transitively (which is allowed in C++), or through C++'s `const` escape hatches. To better understand `const` usage in practice, we ran ConstSanitizer on a benchmark suite and manually classified all writes-through-`const`.

Specifically, the goal of this chapter is to answer the following research questions:

(**RQ3.1**) Do developers perform shallow and transitive writes-through-`const`?

(Answer: Yes to both.)

(**RQ3.2**) How do developers write-through-`const`?

(Answer: By directly writing to fields of `const`-qualified objects and through transitive writes; both are about equally common.)

(**RQ3.3**) Why do developers write to fields of `const`-qualified objects?

(Answer: Buffers and delayed initialization were important reasons, but over half the time, we couldn't find any clear reason motivating developers' decisions to write through `const`.)

Our contributions include:

- the design and implementation of a novel dynamic analysis for C++ that detects writes-through-`const`-qualified variables (both shallow and transitive);
- an empirical study of `const` usage (including writes-through-`const`-qualifiers) on a suite of 7 C++ benchmarks; and,
- based on the empirical study, a novel classification of writes-through-`const`-qualifiers in the wild according to a root cause and a set of attributes.

3.1 Motivation

We continue with an example of a `const` usage that must be accepted by C++ compilers [72] but leads to undefined behaviour when used with the C++11 standard library specification. This may lead to difficult to debug errors.

Listing 3.1 contains function `writeId()`, which modifies field `id` of its parameter. Function `evil()` takes a `const`-qualified parameter, casts away the `const` qualifier, and calls `writeId()`. Both these functions must be accepted by C++ compilers. Recall the purpose of `const_cast` is to work around mis-qualified libraries.

Function `writeId()` does not perform anything unexpected. The write to `id` is legitimate with respect to `const`: `writeId()` has a non-`const` reference to data and is therefore entitled to write to it. At this point the write is valid, no analysis would attempt to prevent it. (`writeId()` conflicts with a different part of the library specification—one oughtn't

Listing 3.1: Method `evil()` violates the spirit of `const` by writing to an externally-visible field of `const` object "`a`". Circled numbers used for subsequent explanations.

```
class A { public: int id; };
size_t std::hash(const A& a) { return std::hash(a.id); }
std::unordered_map<A, std::string> m;
const A a; (1)

m.insert(a, "Value");
evil(a); (2)
m.find(a);                                    (3)

void evil(const A& a) { writeId(const_cast<A*>(&a)); }
void writeId(A *pa) { pa->id = 5; } (4)
```

write to fields used as hashes—but that conflict is beyond the scope of this chapter. We check these kinds of writes later in Chapter 5.)

Function `evil()` accepts a `const` parameter "`a`"; the `const` qualifier intuitively suggests that the state of "`a`" should not change across a call to `evil()`. Chapter 2.2 explains the C++ semantics of `const` in more detail; Section 3.2 explains the writes that ConstSanitizer monitors.) `evil()` then casts away the `const` qualifier and calls `writeId()`, which writes to the `id` field, thus changing the state of "`a`".

Listing 3.1 also contains client code. This client code provides a `hash()` function for when objects of type "`A`" are used with the standard library. It continues with a declaration of an "`std::unordered_map m`", which gets a `const`-qualified object "`a`" added to it. (In C++, objects like "`a`" and `m` are constructed upon declaration.) Between the `insert()` call and the `find()` call, the program calls `evil()`, which changes the `id` field, thus causing the `hash()` of "`a`" to change. As a result, the `find()` call may unexpectedly return `m.end()`, indicating that it did not find "`a`" in the expected bucket; that result should be a surprise to the client.

In our example, the client code is doing nothing wrong, yet it may get an unexpected result from the map. The client should be entitled to believe that its `const`-qualified "`a`" object does not change between the two map calls and that the object is still in the map. However, the client would not find the object in the map after the call to `evil()`.

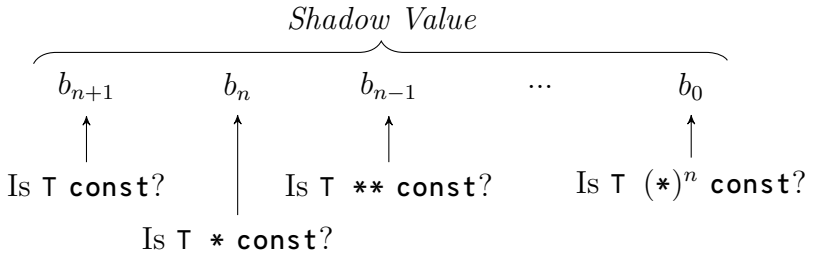


Figure 3.1: Our shadow values encode the `const`-ness of each level of an n level pointer.

Analysis of example. We informally describe how ConstSanitizer works on our example. Our LLVM-based tool actually operates at the `llvm` bitcode level (assisted by metadata from `clang`), but for clarity we describe shadow values and the effects of statements on them using C++ code. Section 3.2 describes our analysis as-implemented on top of LLVM.

ConstSanitizer associates a shadow value with each variable. This shadow value describes whether or not the variable, and each of its dereferences, may be written to; Figure 3.1 shows how shadow values encode `const`-ness. We present the semantics of our shadow encoding more thoroughly in Table 3.1. ConstSanitizer propagates shadow values through the program's execution. At each write, ConstSanitizer verifies that the shadow value permits a write, and indicates a write-through-`const`-qualifier if not.

We next show how ConstSanitizer works; circled numbers refer back to Listing 3.1.

① Program allocates `const`-qualified new object "`a`". Using debug information, ConstSanitizer finds the `const` qualifier in "`a`"'s type, and associates shadow value $(1)_2$ with "`a`", indicating that "`a`" is `const`.

② ConstSanitizer then propagates shadow value $(1)_2$ to the function call `evil()`. Inside function `evil()`, variable "`a`" is a reference, so we shift the shadow value to the left and obtain $(10)_2$ inside the function call boundary.

③ Program casts from type `const A*` to type `A*`. Because "`a`" is a reference inside `evil()`, the address-of operation has no effect on the shadow value. (On a non-reference, taking the address would also result in a logical shift left of the shadow value.)

The cast of the `const` qualifier is invisible to LLVM, so ConstSanitizer propagates shadow value $(10)_2$ across the cast. (Had "`a`" originally not been `const`, a pointer to it would have shadow value $(00)_2$ and it would have shadow value $(0)_2$.)

Finally, ConstSanitizer passes shadow value $(10)_2$ for parameter "`pa`" of `writeId()`.

④ Inside function `writeId()`, the instrumented write to field "`id`" observes that the shadow value of the address of the containing object is $(10)_2$. Because the left-hand side uses the `->` operator, ConstSanitizer shifts the shadow value back right once, giving shadow value $(1)_2$ for the containing object. Because the containing object is `const`, we also apply a shadow value of $(1)_2$ to all writes to fields of that object.

When executing the write, ConstSanitizer checks the right-most bit of the shadow value for the destination. Since this value is $(1)_2$, the program is writing a value through a `const` reference. ConstSanitizer therefore signals a write-through-`const`.

Section 3.3 contains our classification of writes-through-`const`. We classify this write as root cause "C", a write after casting away a `const`, and assign attribute "I", for incorrect. Note that all of the writes-through-`const` we find may not all be writes implementing abstract immutability. We use developer provided `const` qualifiers, this includes all parameters and variable declarations.

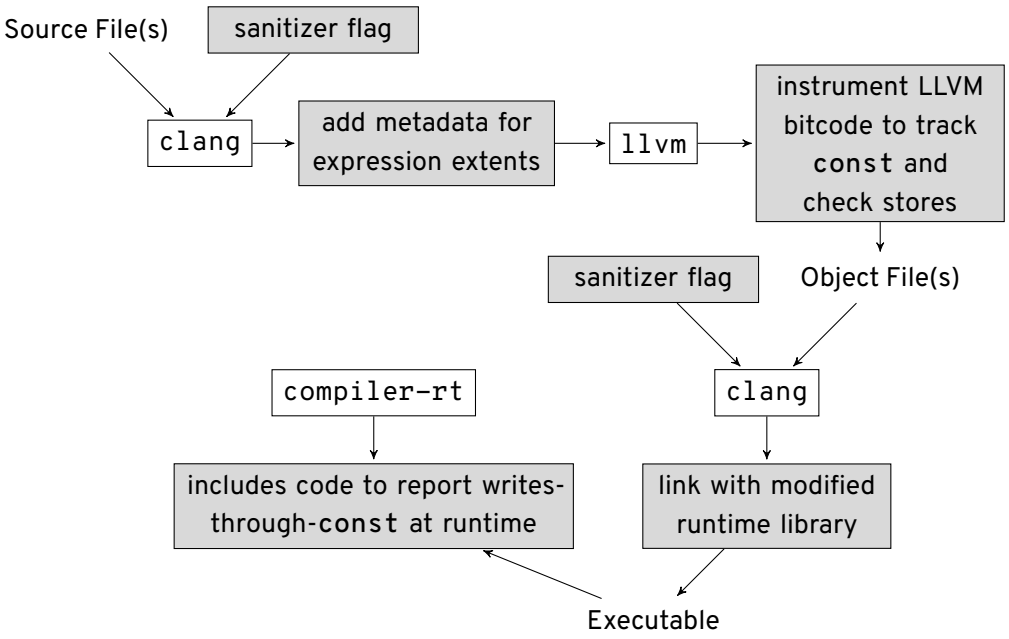


Figure 3.2: ConstSanitizer generates instrumented LLVM bitcode which reports writes through `const` qualifiers at runtime.

3.2 Technique

Our ConstSanitizer tool generates instrumented code which, when executed, prints out notifications about writes-through-`const`-qualifiers. ConstSanitizer builds upon LLVM [80] and was inspired by existing sanitizers including AddressSanitizer [71] and MemorySanitizer [73].

We implemented ConstSanitizer by extending the `clang` frontend and adding instrumentation passes on `llvm` bitcode. The instrumented code calls hooks in our modified version of `llvm`'s `compiler-rt` runtime library. Figure 3.2 depicts our processes for compiling instrumented code. Plain text indicates inputs and outputs; outlined boxes indicate existing software components; and light gray boxes indicate our modifications.

We first describe our modifications to the `clang` frontend. When the developer enables ConstSanitizer (using a command-line flag), our frontend adds metadata about initialization expression extents to the bitcode. This metadata notifies the `llvm`-level instrumentation about source-level constructs that would otherwise be lost in translation to LLVM bitcode.

Specifically, we modified `clang`'s bitcode generator at variable declaration statements. At statements of the form `type var = expr` we mark the instructions making up `expr` so that our `llvm`-level instrumentation can ignore them. The rationale for ignoring those writes is that the primary user-visible write from a `clang` declaration statement is to `var`, on the left-hand side. We empirically observed that other writes within `expr` are almost always initialization writes to `var`, which ought not to be reported even if `var` is `const`. Although a programmer may include explicit (side-effecting) writes within `expr`, we ignore such writes to eliminate the false positives that otherwise occur due to initialization writes.

Listing 3.2: C++ source code showing a false negative due to expression handling.

```
const int * x = new int(0);
int * y = const_cast<int *>(x);
*y = 1; ←—— not reported
```

We use `const`-ness information as provided by declarations, rather than implementing a taint-based approach. Listing 3.2 shows a false negative caused by our approach. The debugging information for `y` gives shadow value $(\texttt{00})_2$. Hence, on the write through `y`, we do not report a write-through-`const`, because we do not propagate `const`-ness information from variable initializers. One might expect this write to trigger a report since `y` aliases the read-only reference `x`. (We report a write if the cast is part of a function argument.)

Most of our instrumentation lives in a custom `llvm` pass that generates code to track `const`-ness of the program's values. The instrumentation manipulates shadow values to track `const` qualifiers at every instruction that generates a pointer value. The `const` information relies on type tables from DWARF 4 debugging information. In `llvm`, this includes all variables in functions—all local variables are allocated on the stack and are pointers.

Our dynamic analysis returns (as one might expect) no false positives, since it observes program executions. However, it depends on the accuracy of the debugging information and metadata, which it uses to identify which variables are `const`-qualified in the source and to identify initialization expression extents. We ran into one false positive in our results which we believe is the result of the metadata being invalidated between LLVM passes. Throughout the remainder of the section, we point out a couple of cases where our analysis must approximate intended `const`-ness because actual `const`-ness information does not exist.

Structure of shadow values. A shadow value consists of n bits tracking `const`-ness (where n is the word length of the processor architecture). Each bit represents whether a

Table 3.1: Shadow values encode available `const`-ness restrictions on variables.

Declaration	Shadow value	Example statement	Allowed
`int x`	$(0)_2$	`x = 5`	✓
`const int x`	$(1)_2$	`x = 5`	×
`int * x`	$(00)_2$	`x = y`	✓
		`*x = 55`	✓
`int *const x`	$(01)_2$	`x = y`	×
		`*x = 55`	✓
`const int * x`	$(10)_2$	`x = y`	✓
(or `int const * x`)		`*x = 55`	×
`const int *const`	$(11)_2$	`x = y`	×
(or `int const *const`)		`*x = 55`	×

pointer or pointee has a `const` qualifier or not. The rightmost bit represents the `const` qualifier of the value itself. Bits to the left (if the value is a pointer) represent what the pointer transitively points to. Our encoding supports pointers up to $n-1$ levels deep on n-bit processors (64 for our experiments, although 32 is sufficient). Figure 3.1 depicted our encoding of shadow values, while Table 3.1 shows how shadow values represent sample `const`-ness settings and corresponding writes allowed. With these examples, a disallowed write will generate a warning with ConstSanitizer.

Shadow value computation. We next describe how we create and propagate shadow values. Our ConstSanitizer instrumentation dynamically propagates shadow values representing `const` qualifiers through a program's instructions. Our goal is to monitor 1) writes to `mutable` fields; 2) locations where `const` has been cast away; and 3) transitive writes, to pointees of fields, through `const` references. Table 3.2 summarizes the analysis rules.

`llvm` bitcode uses `alloca` instructions to introduce new pointer values. Our `llvm` pass instruments each `alloca` instruction with the appropriate shadow value, as extracted from the type information in the source code, using standard `clang` debug information.

Ultimately, our instrumentation verifies the behaviour of `store` instructions. Recall that we exclude `store` instructions that come from the right-hand side of a declaration statement. For all other `store`s, we check whether the operand—the location being written-to—represents a `const`-qualified type. The rightmost bit of the shadow value provides this information. If that bit is 1, an execution of this `store` instruction is a write to a `const`-qualified location. We insert a call to our runtime library to check the value of the bit and to report a write-through-`const` if the bit is 1. (We later discuss a special case for store instructions where the value being stored is a function argument.)

Conversely, `load` instructions return a pointer that represents a single pointer dereference. To compute the returned shadow value, we right shift the operand's shadow value. Recall that the least significant bit represents the current `const`-qualifier applied to the value.

`llvm`'s `getelementptr` (GEP) instruction accesses arrays and fields of objects. This instruction preserves type safety through dereferences in the compilation process and is a safe alternative to directly generating pointer arithmetic code. Our instrumentation performs a logical shift right by one bit for every pointer dereference implied in the GEP instruction. Our treatment of GEP implicitly handles transitive immutability as follows: when a GEP accesses an object field, and the containing object is `const`-qualified, we generate a shadow value as if the field had a `const` qualifer on every type for the contained field (this corresponds to a shadow value of all 1s). This treatment implies checks for transitive immutability; generating a shadow value of 1 here (only the least significant bit is 1) would generate the same bitwise immutability checks for `const` as specified by the C++ standard.

Our instrumentation propagates `const`-ness information (in shadow values) alongside references to that location. In C++, access restrictions to a location depend on whether the program is accessing that location through a `const` reference or not. Therefore, in the presence of casts and pointer arithmetic, there is no ground truth about the `const`-ness of the resulting references and we must make a reasonable under-approximation as to

`const`-ness.

We next discuss casting-related `llvm` instructions. The `bitcast` instruction converts a value into a specified type. If a program converts a pointer between equally-indirected pointer types, then we copy the old shadow value to the result. (C++ `const`-casts do not appear at LLVM bitcode level, nor do the component of a C-style cast that manipulates `const`-ness. ConstSanitizer preserves declared `const`-ness for such variables.) Otherwise, we choose to assume that the instruction's result has no `const` qualifiers. We make this assumption in all cases for the `inttoptr` instruction, which represents pointer arithmetic not handled by the GEP instruction, as well as for `extractvalue` and `extractelement`.

Our instrumentation stores shadow values for function calls' arguments and return

Table 3.2: Dynamic analysis rules showing computation of shadow value for result `%1`.

Instruction	**New shadow value**
`%1 = alloca ...`	from `const` qualifiers in debugging information, consistent with Figure 3.1.
`%1 = getelementptr %2`	by logically shifting right `%2`'s shadow value once for each dereference this instruction represents. **if field access**: check `const` qualifier of base object; for (immutable) `const` base objects, new shadow value is all ones, otherwise all zeros.
`%1 = call(%2)`	loaded from return shadow value in Thread-Local Storage (TLS). **for pointer arguments `%2`:** also write shadow values to appropriate TLS slots for the function call; if the call and argument are marked as ignored, write all zeros for the shadow value for the argument.
`%1 = phi/select ...`	carry out same operation on shadow value operands.
`%1 = bitcast %2`	from the shadow value for `%2`, if compatible; otherwise all zeros.
`%1 = load %2`	logical shift right of `%2`'s shadow value.
`store %2, %1`	check rightmost bit of shadow value for `%1`, report write-through-`const` if set. (Only applies if the instruction not ignored as an initializer.) **if `%2` is function argument:** load shadow value for `%2` from TLS, left shifted once. New shadow value is bitwise OR of shifted value with previously computed shadow value for `%1`. **if `%2` is "this" function argument:** same steps as above, except skip the bitwise OR step. **if `%2` is "this" function argument for destructor:** shadow value for `%1` is $(00)_2$.

`%1 = extractelement`
`%1 = extractvalue`
`%1 = inttoptr`
`%1 = landingpad`
	all zeros.

values using thread local storage (TLS). In the straightforward case, we store shadow values for pointer arguments in TLS slots reserved for each argument. However, we ignore pointer arguments' `const` qualifiers if the call and the argument are both part of a variable declaration. As for (pointer) return values: if a function was instrumented by our tool, then we read the shadow value from the appropriate TLS slot. We also store a mutable shadow value in the return value TLS slot, in case the function had not been instrumented by our tool. Our instrumentation either reads the approximation or, if applicable, the actual return value generated by the called function. In the presence of callbacks from uninstrumented code back to instrumented code, our instrumentation may use stale shadow values and report extraneous results based on these stale shadow values.

We have a special case for store instructions where the value stored is a function argument, as mentioned above. Consider a store instruction "`store value, location`". We compute the shadow value for "`location`" as follows. First, we get the shadow value for "`value`" from the TLS. Then, we adjust this shadow value to be compatible with the type of "`location`": our encoding requires one logical shift to match the type of "`value`" to that of "`location`". We bitwise OR the shadow value for "`location`" previously computed (from an `alloca` instruction) with the shifted value to get the new shadow value for "`location`". This preserves `const` qualifiers of the original argument and of the local variables in the function.

There are two further special sub-cases for store instructions and function arguments for (i) method and (ii) destructor calls. Listing 3.3 illustrates sub-case (i). Here, `foo()` is declared `const`. The compiler will hence treat `this` as `const` within `foo()`. However, for our dynamic analysis, we want to detect writes based on the `const`-ness of `this` from the caller; in method `bar()` in Listing 3.3, receiver object `nc` for `foo` is not `const`, so we do not want to report the call's (transitive) store to `x`. `foo`'s method arguments appear as "`value`" operands of `store` instructions while the "`location`" is an `alloca` within the function. We set the shadow value of the associated `alloca` instruction to the value of the argument after applying a logical shift left by one (since it's a pointer). This treatment properly ignores `const` qualifiers added due to callee method signatures.

For sub-case (ii), destructors, we do not want to report any writes through `this` as the object no longer exists after the call (so that writes to the object aren't visible in any case). We handle this case by simply assuming that the `this` argument is mutable.

For all other arguments, we do a bitwise OR between the `alloca` shadow value and the argument shadow value logically shifted left by one, which maintains all `const` qualifiers. This ensures that if either the original variable type or the argument type has a `const` qualifier we would report a write-through-`const`.

Shadow value computation example. Listing 3.3 presents the C++ source code for `C::foo`, a `const` qualified method, and the associated LLVM bitcode. Consider the `bar` function, which calls `foo` twice, first with mutable (i.e. non-`const`) receiver object `nc` and then with `const` receiver object `cc`. Within `bar`, the shadow value of `nc` is $(0)_2$ and the shadow value of `cc` is $(1)_2$. Our instrumentation assigns shadow values for each LLVM instruction with a pointer result. We instrument `C::foo` as follows:

- The first instruction, `alloca`, stores its result in `%1`. Since it is an `alloca` instruc-

Listing 3.3: C++ source code showing calls to method `foo()` (with its definition and associated LLVM bitcode) from `const` context `cc` and non-`const` context `nc`.

```
void bar() {
  C nc;
  const C cc;
  nc.foo();
  cc.foo();
}

class C {
  int *x;
public:
  void foo() const {
    *x = 42;
  }
};

define void @C::foo(%class.C* %this) {
  %1 = alloca %class.C*
  store %class.C* %this, %class.C** %1
  %2 = load %class.C*, %class.C** %1
  %3 = getelementptr %class.C, %class.C*
                     %2, i32 0, i32 0
  %4 = load i32*, i32** %3
  store i32 42, i32* %4
  ret void
}
```

foo's definition

foo's LLVM bitcode

tion, we obtain its shadow value from `clang` debugging information. The associated shadow value is $(10)_2$: in this `const`-qualified method, the type of `this` is that of a `const` pointer to the containing class, `const C *`.

- At the `store` instruction, without special handling, we would load the shadow value of argument `%this` from the TLS; logically shift left the shadow value by one to account for the fact that we are performing a `store` to memory allocated for that argument; and bitwise OR the resulting shadow value with the original shadow value for `%1`. In our example, whether the receiver object is `cc` or `nc`, the shadow value for `%1` is $(10)_2$.

- Next, we obtain the shadow value for the result of the `load`, `%2`. As `%2` returns a pointer, we shift `%1`'s (the operand's) shadow value right by one, giving a shadow value of $(1)_2$.

- Next, the `getelementptr` instruction results in a pointer to the class's `x` field. Our instrumentation of `getelementptr` could produce two different shadow values, depending on the instruction's operand. In this case, `%2` is a `const` object, and the resulting shadow value for a fully `const`-qualified `x` field is $(11)_2$.

- Next, we obtain the shadow value for the `load` result `%4` using the same technique as for `%2`. The resulting shadow value is $(1)_2$.

- Finally, we insert a check at the `store` instruction. In this case, the least significant bit of the shadow value associated with location (`%4`) is 1. Therefore we would dynamically report a write-through-`const` at the write to field `x` of the `const` method.

For methods, this instrumentation is not enough. We only want to report a write-through-`const` for the call with `const` receiver object even though both objects call the same static method. Before the call to `foo`, our instrumentation stores the shadow value of the receiver object in its TLS slot. Our instrumentation of `foo` looks for `store` instructions that use the receiver object and recomputes the shadow value of the location. Here, we load the shadow value from its TLS slot and shift left to match the type of the expected shadow value of `%1`. For `nc`'s call, this shadow value is $(00)_2$. Since `foo` is a method, we ignore the original shadow value of `%1` ($(10)_2$) and overwrite it with new shadow value $(00)_2$. Following the remaining steps in `foo` as above, the shadow value of `%4` is now $(0)_2$ and we do not report a write-through-`const`. In the `cc` case, we would follow the same steps, but instead report a write-through-`const`, because the shadow value would be $(10)_2$.

3.3 Classification

One of our contributions is a careful analysis of the `const` usages detected by our ConstSanitizer dynamic analysis tool. We propose a classification for writes-through-`const`-qualifiers along 2 axes. We manually assigned each write 1) a single cause, from a set of common root causes; and 2) a set of additional attributes. This classification distills our empirical observations about `const` use in practice.

Table 3.3: Root causes of writes through `const` and our symbols for these causes.

Root Cause	Symbol
Write to mutable field	M
Transitive write	T
Write after casting a `const` qualifier away	C

Table 3.3 lists all of the root causes for writes-through-`const`, along with a one-letter abbreviation that we will use in Section 3.4's tables. ConstSanitizer detects such writes and reports them to the user. The causes are:

- mutable field (M): the program writes to a `mutable`-labelled field of a `const` object.

```
class Mutable {
  mutable int x;
public:
  void mutator() const { x = 42; }
};
```

  `mutable` permits method `mutator()` to write to field `x` even though it is a `const` method, which would ordinarily prevent (at compile-time) writes to fields of the `this` object.

- transitive write (T): the program writes through a field of a `const` object.

```
class TW {
  int *x;
public:
  void transitiveWrite() const { *x = 42; }
};
```

  `const`-qualified method `transitiveWrite()` writes through field `x` of the `this` object. While the `const` qualifier prevents mutation of the `x` field, it does not prevent transitive writes of the memory pointed to by `x`.

- casting away const (C): the program writes through a pointer which has previously been `const` but whose `const`-ness has been cast away using a `const_cast` or C-style cast.

```
void writeToArg(int *y) { *y = 17; }

const int *x = ...;
writeToArg(const_cast<int *>(x));
```

  The write in `writeToArg()` mutates the value pointed-to by `x` while `x` is `const`-qualified. ConstSanitizer reports writes-through-`const`-qualifiers whose `const`-ness has been cast away, using the `const`-ness of the most recent declared type for the value.

Table 3.4: Observed attributes of writes through `const` and corresponding symbols.

Attribute	Symbol
Write is synchronized	S
Write is not visible	N
Write is to a buffer/cache	B
Write is delayed initialization	D
Write is incorrect	I

Table 3.4 summarizes our attributes for writes-through-`const`-qualifiers. We assigned attributes to writes based on our understanding of the code. Writes may have multiple attributes; for instance, a write in our Protobuf benchmark is B & N & S. The attributes are:

- synchronized (S): indicates that the write is always protected by a lock. This attribute is often required under the C++11 standard: all types that are shared between threads and that may be used with the standard library must be either bitwise const, which is clearly not the case when we witness a write, or else protected against concurrent accesses [76]. The following example, from Protobuf, is synchronized using Google mutex primitives.

```
GOOGLE_SAFE_CONCURRENT_WRITES_BEGIN();
_cached_size_ = total_size;
GOOGLE_SAFE_CONCURRENT_WRITES_END();
```

- not visible (N): indicates that the result of the write is never externally visible (e.g. private and with no accessor methods; may be accessed in the same translation unit). Often occurs in the context of testing-related counters.

```
mutable int countFooCalls;
void foo() const { ++countFooCalls; }
```

- to a buffer/cache (B): indicates that the write is of a derived value which can be computed from other currently-available state. Such writes are often optimizations.

```
_cached_size_ = total_size;
```

- delayed initialization (D): indicates that the write initializes state not initialized in the constructor or its transitive callees. Writes with this attribute could have also occurred in the constructor, but the written value was not yet available. Failure to call a delayed initialization method would lead to undesired behaviours (or lack of desired behaviours).

```
bool Generator::Generate(const FileDescriptor* file, ...) const {
  this->file_ = file;
}
```

- incorrect (I): indicates that the write appeared to violate the `const`-ness of the object.

Note that S/N/B/D writes are not necessarily errors and do not necessarily violate immutability properties. We thus chose the word "attribute" to suggest that S/N/B/D indicate an incidental property of a write-through-`const`. If the code containing the writes is properly written, an object with an S/N/B/D write-through-`const` can still appear to be immutable to the client, assuming all references to that object are read-only. A write with attribute I, however, is a client-visible violation of `const`.

3.4 Results

We evaluated our ConstSanitizer tool on 7 C++ software projects, plus 1 C project. We attempted to choose significant benchmarks using these guidelines:

1. must span a range of application areas: applications and libraries; small, medium, and large projects; interactive and non-interactive;

2. are used by the community: the Google projects are the most popular on GitHub; the applications are popular among FOSS users; contributor-group sizes vary from a core group to a large community; and,

3. must extensively use `const` constructs.

A ConstSanitizer report indicates that a write that would not be allowed under deep immutability occurred through a read-only reference. Such writes are allowed under C++ semantics. They are only a departure from the `const` semantics that we experiment with (i.e. deep immutability with no casts and no mutable). Our experiments classify writes-through-`const` observed in actual `const`-using programs. Classifying these writes provides us valuable insight about `const` usage in practice, which will guide future work.

Our approach was to modify the project's build system to use our tool and to disable optimizations. We then ran the project's test suite, when available, and collected output from our instrumentation. Using this output we categorized the writes that we found, assigning root causes and attributes. Along with the number of static locations of writes that we found (**bolded**), we also report the number of dynamic occurrences of each write over observed executions. All else being equal, dynamic counts can help prioritize writes-through-`const`, with more-frequent locations to be investigated first. We refer to these dynamic occurrences of writes as "occurrences" in the sequel. Table 3.5 summarizes our benchmark projects.

We recorded relative overhead introduced by our instrumentation with respect to both building and testing times on the longest-running projects, Protobuf and LevelDB. Table 3.5 includes build slowdowns induced by our tool, which ranged between 1.05× and 1.40×. Our tool caused a 3.3× slowdown and 1.3× slowdown in test execution times for Protobuf and LevelDB respectively. The remaining projects were either interactive, or did not have long enough running test suites to get meaningful results. We do not report any LLVM TableGen numbers because we built it (with instrumentation) as part of the LLVM build process and were not able to build the LLVM TablGen executable separately.

Table 3.5: We ran our experiments across 7 C++ (and 1 C) software projects; ConstSanitizer introduces a build slowdown of 1.05×–1.40× across all projects.

	Name	Version	Description	Build Slowdown
C++	Protobuf	2.6.1	Serialization framework	1.40×
C++	LevelDB	1.18	Key/value database	1.05×
C++	fish shell	2.2.0	UNIX shell	1.32×
C++	Mosh (mobile shell)	1.2.5	SSH replacement	1.26×
C++	LLVM TableGen	3.7.0	Domain-specific generator	—
C++	Tesseract	3.04.00	OCR engine	1.10×
C++	Ninja	1.6.0	Build system	1.20×
C	Weston	1.9.0	Wayland compositor	1.28×

3.4.1 Protobuf

Protobuf is Google's serializing framework for structured data, consisting of about 214 000 (we do not count comments or whitespace) lines of C++ code. We analyzed version 2.6.1 of Protobuf by running its test suite, which contains 5 tests. Table 3.6 summarizes the Protobuf results. ConstSanitizer found **76** static write locations (and 127 644 occurrences). We describe 5 archetypes for these writes. An archetype is a group of writes that we judged to be similar; the writes may happen at different source locations.

Table 3.6: Protobuf has 5 archetypes from **76** writes-through-`const` (127 644 occurrences).

Archetype	Locations	Occurrences	Root Cause	Attributes
Generator printer	**7**	118464	T	B & N & S
Message cache sizes	**61**	7158	M	B & S
Source code locations	**4**	1898	T	I
Linked list operations	**2**	84	M	I & S
Generate initialization method	**2**	40	M	D & N & S

The "Generator printer" archetype occurred most often. Listing 3.4 presents a representative expanded stack trace. The function at the top of the listing shows the initiation of the write in `Generator`'s `const`-qualified `Generate` method. This method calls `PrintTopBoilerplate`, passing a pointer to a mutable `io::Printer`. Then, `Printer`'s `WriteRaw` method modifies (root cause T) two fields: `buffer_` and `buffer_size_`. These fields are protected by a lock, act as a buffer, and are not visible outside the class (which is just a printer). This archetype also includes other `Print`-like calls with different source locations but a common explanation.

The "Generate initialization method" archetype is related to "Generator printer". Listing 3.5 shows this archetype. The `printer_` field was initialized as seen above. C++ allows this write due to the `mutable` specifier. Another field, `file_`, is lazily initialized as well.

Listing 3.4: Protobuf's Generator class performing transitive write-through-`const` to a Printer field.

```
bool Generator::Generate(...) const {
  PrintTopBoilerplate(this->printer_, ...);
}

void PrintTopBoilerplate(io::Printer* printer, ...) {
  printer->Print(...);
}
```

python_generator.cc

```
void Printer::Print(...) {
  WriteRaw(text + pos, i - pos + 1);
}

void Printer::WriteRaw(..., int size)
  this->buffer_ += size;
  this->buffer_size_ -= size;
}
```

printer.cc

transitive writes initiated by `const ::Generate()`

Both of these fields are protected by the same lock, and are not externally visible outside the class.

Listing 3.5: Protobuf's Generate initialization method performing lazy initialization.

```
bool Generator::Generate(...) const {
  this->file_ = file;
  this->printer_ = &printer;
}
```

python_generator.cc

We show an example of the "linked list operations" archetype in Listing 3.6. Here, the `depart` method grabs a lock, and uses a pointer with type `linked_ptr_internal const *`, so that the `const` applies to what is pointed to, not to the pointer. The method then modifies the `next_` field of a valid object at the point indicated by the comment. The root cause here is `mutable`: the `next_` field is declared `mutable linked_ptr_internal const* next_`. This write, in the `depart()` method, is an incorrectly labelled write through `p`, and synchronized.

Listing 3.6: Protobuf using Google test linked list that writes internally.

```
bool linked_ptr_internal::depart()
    GTEST_LOCK_EXCLUDED_(g_linked_ptr_mutex) {
  MutexLock lock(&g_linked_ptr_mutex);

  if (this->next_ == this) return true;
  linked_ptr_internal const* p = this->next_;
  while (p->next_ != this) p = p->next_;
```

Listing 3.8: Protobuf writing to a source location object.

```
void Parser::LocationRecorder::AttachComments(...) const {
  this->location_->mutable_leading_comments()->swap(*leading);
}
```
parser.cc

descriptor.pb.h

```
std::string* SourceCodeInfo_Location::mutable_leading_comments() {
  this->leading_comments_ = new ::std::string;
}
```

```
  p->next_ = this->next_;
  return false;
}
```

Listing 3.7 shows the "Message cache sizes" archetype. The write is protected by a lock, and is allowed by C++ because the field is `mutable`. However, this write, while involved with caching, is externally visible. The method `void SetCachedSize(int size) const` enables external code to modify this field through a `const` reference to the containing object.

Listing 3.7: Protobuf writing to a message's cached size field.

descriptor.pb.cc

```
int FieldDescriptorProto::ByteSize() const {
  GOOGLE_SAFE_CONCURRENT_WRITES_BEGIN();
  this->_cached_size_ = total_size;
  GOOGLE_SAFE_CONCURRENT_WRITES_END();
}
```

Listing 3.8 shows the "Source code locations" archetype we found in Protobuf. The `mutable_leading_comments` method, which includes "mutable" in its name, is not declared as `const`, and thus allows writes. Its implementation writes to the `location_` field; we show an example of a caller which causes such a write. The `location_` field is externally-visible, so this is a clearly incorrect externally-visible transitive write; we assign attribute I.

We also found an archetype involving writing data to a message. This included 133 unique source locations, occurring 14 638 times in total. However, the code is heavily inlined and the build system appears to overwrite optimization settings for this subdirectory. Manual inspection of the code revealed no obvious writes. We believe this is a result of optimizations causing invalid debugging information. We thus omitted this archetype from Table 3.6.

3.4.2 LevelDB

LevelDB (1.18) is Google's lightweight key/value database library, consisting of approximately 18 000 lines of C++ code. The test suite contains 23 test drivers. There were 6

archetypes and also **6** root source locations for these writes. These locations contributed to 13 792 occurrences over the test drivers. Table 3.7 shows a summary of our findings for LevelDB.

Table 3.7: LevelDB shows writes from **6** source locations, with 13 792 occurrences in total.

Location	Occurrences	Root Cause	Attributes
`db/db_test.cc:40`	10311	T	N & S
`util/cache.cc:315`	2841	T	B & S
`db/snapshot.h:54`	319	T	I
`db/snapshot.h:55`	319	T	I
`helpers/memenv/memenv.cc:274`	1	T	I & S
`util/testutil.h:42`	1	T	N

Listing 3.9 shows the source location that caused the majority of the occurrences. This code extends the `RandomAccessFile` class to add an atomic counter field, `counter_`, that tracks the number of read calls. The root cause is that `counter_` is a pointer and is transitively written to. The reason for this write is test controllability: this class is part of the test infrastructure. Yet it must override the monitored call (and thus must be `const`). This class is meant for testing purposes only, so we concluded the write was not visible outside the class—the counter is only used in the testing code.

Listing 3.9: LevelDB write in `db_test.cc` incrementing counter tracking # of writes to a file.

db_test.cc

```
class CountingFile : public RandomAccessFile {
  virtual Status Read(...) const {
    this->counter_->Increment();
  }
};
```

Listing 3.10 shows a modification to a caching structure that generates a new identifier. This cache is a field, `block_cache`, in `options`, which is declared as `const Options&` in `Table::Open`. The root cause is a transitive write, since the code dereferences a field of a `const` object to do the write. This write is protected by a lock and clearly involved in caching. However, it appears that other code outside of `Options` uses this block cache field.

Listing 3.11 shows a modification of a linked list node accessed through two pointer dereferences. This corresponds to both `snapshot.h` locations shown in the table. This code modifies the pointers obtained from following its own nodes, performing a transitive write through a `const` qualifier. We do not know why the developers declared `s` as `const` since it is also destroyed at the end of the method. In any case, we assigned this write attribute I.

Listing 3.12 shows a write-through-`const` in the `InMemoryEnv` class. As with the cache, the root cause is a transitive write: in the caller, `options` is declared `const Options&`.

Listing 3.10: LevelDB write in `cache.cc` creating a new block cache in `const Options` object.

```
Status Table::Open(const Options& options, ...) {
    rep->cache_id = (options.block_cache ?
                     options.block_cache->NewId() : 0);
}
```

table.cc

```
virtual ShardedLRUCache::uint64_t NewId() {
  MutexLock l(&id_mutex_);
  return ++(last_id_);
}
```

cache.cc

Listing 3.11: LevelDB write in `snapshot.h` deleting a list element and updates pointers.

```
void Delete(const SnapshotImpl* s) {
  assert(s->list_ == this);
  s->prev_->next_ = s->next_;
  s->next_->prev_ = s->prev_;
  delete s;
}
```

snapshot.h

Unlike the caching example, this file isn't involved in caching and appears to be a visible change to `options`. This write in `NewWritableFile()` is protected by a lock, giving attribute I & S.

Listing 3.13 shows the final write-through-`const`-qualifier that we found for LevelDB. The caller location is the same as in Listing 3.12 above. In this case, however, the containing class extends `InMemoryEnv` and adds a field to count the number of errors (for testing purposes only). Therefore we attribute this write as being not visible—it is only used in tests.

Listing 3.12: LevelDB write in `memenv.cc` changing the environment in options object.

```
Status DB::Open(const Options& options, ...) {
  s = options.env->NewWritableFile(...);
}
```

db_impl.cc

```
... InMemoryEnv::NewWritableFile(...) {
  MutexLock lock(&mutex_);
  file_map_[fname] = file;
}
```

memenv.cc

Listing 3.13: LevelDB write in `testutil.h` injecting faults into the test suite.

```
... EnvError::NewWritableFile(...) {                    testutil.h
  ++this->num_writable_file_errors_;
}
```

3.4.3 fish shell

fish shell (2.2.0) is a UNIX shell providing advanced features, consisting of approximately 48 000 lines of C++ code. We compiled the project with our tool and executed an instance of the shell. Our workload launched the shell and immediately exited. We found writes from **4** unique source locations for 98 occurrences in total. All locations are within the `exchange` function. Listing 3.14 shows this function along with a snippet of `_wgetopt_internal` that calls `exchange`. The root cause is that the `const`-qualified `argv` variable gets cast to non-`const` and then passed to `exchange`. This write shows that the `const`-qualifier on `argv` is incorrect and should not be included.

Listing 3.14: fish shell writing to `const`-qualified `argv` object.

```
... _wgetopt_internal(..., wchar_t *const *argv, ...) {        wgetopt.cpp
  exchange((wchar_t **) argv);
}

... exchange(wchar_t **argv) {
    argv[bottom + i] = argv[top - (middle - bottom) + i];
    argv[top - (middle - bottom) + i] = tem;
    argv[bottom + i] = argv[middle + i];
    argv[middle + i] = tem;
}
```

3.4.4 Mosh (mobile shell)

Mosh (mobile shell) (1.2.5) is a remote terminal application that is a replacement for secure shell (SSH), consisting of about 13 000 lines of C++ code. Our workload was to launch the mosh server and immediately terminate it. We found writes-through-`const` at **8** unique source locations (432 occurrences). Listing 3.15 shows one of the writes. Mosh parsing code sets a flag to indicate completion. However, the developers declared the parser action as `const` in the same method where they modify it. The root cause is that the variable `handled` is declared `public mutable`. We believe this is an incorrectly `const` qualified variable.

Listing 3.15: Mosh handling terminal action with a write-through-`const`.

```
void Emulator::print(const Parser::Print *act) {        terminal.cc
  act->handled = true;
}
```

Listing 3.17: LLVM SubReg writes to a `mutable` field in a `const` method.

```
unsigned CodeGenSubRegIndex::computeLaneMask() const { CodeGenRegisters.cpp
  if (this->LaneMask)
    return this->LaneMask;
  this->LaneMask = ~0u;
  unsigned M = ...;
  this->LaneMask = M;
  return this->LaneMask;
}
```

3.4.5 LLVM TableGen

We instrumented LLVM's (3.7) TableGen executable, which uses domain-specific information to generate files with custom backends. This part of LLVM consists of approximately 34 400 lines of C++ code. It is primarily used in building LLVM itself. We added our instrumentation to the build system and observed an instrumented version of TableGen executing as part of the build process. LLVM itself is a large body of code with too many writes-through-`const`-qualifier objects to manually classify. In TableGen, we found writes from **3** unique source locations (282 occurrences).

The handling code for DFAs contains some puzzling writes, shown in Listing 3.16. The write immediately follows an instantiation of a `const State` object. The `State` class itself is only available in a file's translation unit (not usable outside the file), which may indicate that the `State` is not intended to be widely used. `State` only contains `const` methods and all of its fields (except one explicitly declared `const`) are mutable. Since all methods are `const` there is no difference in callable methods between non-`const` and `const`-qualified access. In addition, since all other fields are mutable, developers are allowed to re-assign the same fields in a `const` method as they would in a non-`const` method. Since only one field doesn't have `mutable`, developers could achieve the same effect by making all methods non-`const`, removing all `mutable` specifiers on fields, and changing the one field that did not have `mutable` to be `const` qualified. From inspection, this is not an immutable object. We found an `addTransition` method that was `const`-qualified, which clear mutates the object.

Listing 3.16: LLVM DFA code marks `State const` for no apparent reason.

```
void DFAPacketizerEmitter::run(raw_ostream &OS) {  DFAPacketizerEmitter.cpp
  const State *NewState;
  NewState = &D.newState();
  NewState->stateInfo = NewStateResources;
}
```

The other write is in the code that computes a sub-register index for code generation. Listing 3.17 shows the containing method. The root cause here is that the `LaneMask` field is mutable. The write caches the value. However, this value is not used in any other methods.

3.4.6 Tesseract

Tesseract (3.04.00) is an optical character recognition (OCR) engine maintained by Google, consisting of 147 000 lines of C++ code. This project does not contain any easy-to-run tests. We compiled it with our tool and ran it with invalid arguments. With our limited knowledge of Tesseract's usage, we were not able to cause the core algorithm to execute. However, we found a strange write, shown in Listing 3.18. The root cause is that the `used_` field is mutable. This write appears to be an incorrect usage of `const`. Strangely, however, the comments indicate that is a defensive write against possible further writes-through-`const`-qualifiers.

Listing 3.18: Tesseract performs a strange write in its string class.

```
const char* STRING::string() const {                    strngs.cpp
  const STRING_HEADER* header = GetHeader();
  header->used_ = -1;
  return GetCStr();
}
```

```
/* mark header length unreliable because tesseract might
cast away the const and mutate the string directly. */
```

3.4.7 Ninja

Ninja (1.6.0) is a build system consisting of approximately 14 900 lines of C++ code. It includes a modest test suite. Our tool reports 39 occurrences from calls to the standard library. All of these warnings have a **single** source location outside the standard library: `src/disk_interface_test.cc:226:3`. Listing 3.19 shows this static source location. This is a quick hack to run the test suite with the same API as normal clients. This field stores statistics that are checked in the test suite only. The field is mutable and not seen outside the test suite, so we give this write the "not visible" attribute.

Listing 3.19: Ninja write-through-`const` in test code.

```
TimeStamp StatTest::Stat(const string& path, ...) const {
  this->stats_.push_back(path);
}                                                disk_interface_test.cc
```

3.4.8 Weston

While we focus on C++ in this work, our technique also works on `const` in C programs. We therefore evaluated it on a C application. Since this is the sole C project, we omit Weston from the overall table of results (Table 3.8). Weston (1.9.0) is a reference implementation of a Wayland compositor. It consists of approximately 85 000 lines of C code and the test suite has 20 tests. We did not expect to see many writes, as most C standard library functions do not require `const` (corresponding C++ library functions usually do), and also due to annoyances in using `const` in C, which we describe below. However, even with a small test suite, we found **4** unique source locations for writes (accounting for 115 occurrences).

All of the writes-through-`const` are transitive and came from parsing code. The argument option parser accounts for **3** locations. Listing 3.20 shows a write in the parser. Function `handle_option` does not modify the pointer value of `option` but modifies its transitive `data` field. This does not change any data stored in the `weston_option` structure, maintaining bitwise `const`-ness. The `data` field's type is `void *` and the cast does not remove `const`. Based on the function name, one might expect a write to the `data` field, not its pointee.

Listing 3.20: Weston option parser modifying its `const` option argument.

option-parser.c
```
handle_option(const struct weston_option *option, ...) {
  * (char **) option->data = strdup(value);
}
```

The final location was in the configuration file parsing code. Listing 3.21 shows the `weston_config_section_get_uint` function dereferencing and modifying the `value` argument passed in from a field of a `const` struct. As above, based on the function naming, one would expect any writes to happen through the `dest` pointer. This write does not modify any data stored in the `config_command` structure and maintains bitwise `const`-ness as well.

Listing 3.21: Weston config parser writing to its `value` argument.

hmi-controller.c
```
struct config_command {
  char *key;
  uint32_t *dest;
};
const struct config_command *command = ...;

weston_config_section_get_uint(..., command->dest, ...);

... weston_config_section_get_uint(..., uint32_t *value, ...) {
    *value = strtoul(entry->value, &end, 0);
}
```

We made an observation as to why `const` may be unattractive to C developers: there is no clean way to initialize a structure analogous to C++ constructors/destructors. A popular C idiom is to assign constructor-like functions signatures like `rec_init(struct rec *r)`. This signature prevents initialization without casting: `const struct rec r; rec_init(&r)` is illegal. However, it is cumbersome to always cast for constructor-like calls. One could change the signature of the function to `rec_init(const struct rec *r)` and perform the cast in the function. However, that function would violate shallow immutability—it writes to fields as it initializes them. Using `const` in C appears to require developers to ignore the casting away of `const` qualifiers for constructor-like functions.

3.4.9 Summary

Table 3.8 summarizes the writes-through-`const`-qualifiers from benchmarks other than Protobuf and LevelDB. Across the 7 C++ projects we instrumented and ran, we observed 17 unique archetypes across a total of 142 288 dynamic occurrences. We manually divided these archetypes into 17 classifications. The root causes were evenly split between writes through mutable fields and transitive writes (8 of each) with one write-through-`const` due to casting. Valid attributes were mostly with-synchronization and because the write was not visible (7 and 6 respectively). The other valid attributes, writing to a buffer/cache and delayed initialization, occurred 4 times and 1 time respectively. The majority attribute, in 9 cases, was that the write was incorrect and violated intuitive notions of what `const` should mean. We reported our results to developers. Within a few days the developers simply removed incorrect `const` qualifiers in both fish and Mosh.

Table 3.8: Writes-through-`const`-qualifiers in other benchmark programs were mainly incorrect uses of `const`.

Project	Location	Occurrences	Root Cause	Attributes
fish shell	`wgetopt.cpp`	98	C	I
Mosh	`terminal.cc`	432	M	I
LLVM	`DFAPacketizerEmiter.cpp`	112	M	I
TableGen	`CodeGenRegisters.cpp`	170	M	B & N
Tesseract	`ccutil/strngs.cpp`	1	M	I
Ninja	`disk_interface_test.cpp`	39	M	N

We found 3 projects (LevelDB, Mosh, and Ninja) had writes for the purposes of testing. The writes-through-`const` we found in testing code were writes to counters only present in test environments. In Mosh, the fact that the writes were only for test purposes was not immediately obvious. However, discussions with the developers revealed that the `handled` variable was only used for debugging. All of these writes-through-`const` are related to test controllability, suggesting that this idiom should be supported directly in the programming language.

Since this is a dynamic analysis, we only observe writes-through-`const` at run-time. These results do not show how often developers use `const` more broadly. If developers use `const` only in these complex cases, developers could manually review all cases of `const` to check correctness. In this case developers would not find writes-through-`const` unexpected if that is their sole use case.

Chapter 4

Static Observations of Immutability

Modern programming languages enable developers to declare that objects (or parts thereof) are immutable. Immutability enables both developers and compilers to better reason about the code and potentially enables compiler optimizations. For instance, objects without mutable state are immune from unexpected changes to mutable state, thus simplifying debugging and program maintenance. Furthermore, immutable objects can be shared between concurrent threads of execution without the need for locking, thereby helping enable automatic parallelization. One of our goals was to understand whether developers buy into arguments in favour of immutability and actually label immutability in their code.

In the previous chapter we found, dynamically, that developers violate deep concrete immutability through immutability declarations. This chapter, by contrast, examines developers' API design decisions and when developers label members as `const`. We investigate developers' immutability declarations (through `const`) more broadly. If we find that a simpler analysis cannot check developer's `const` declarations it justifies the need for a more complex abstract immutability analysis.

C++ developers are encouraged to use the `const` keyword to declare immutability [52]. In this work, we statically investigate how C++ developers use `const` in classes' public interfaces. For instance, are there many classes without mutable state? More generally, we wanted to know which fields and methods C++ developers `const`-annotated, and why.

We developed a tool that identifies immutable classes (where all members are immutable) as well as all-mutating classes (where no members are declared immutable). Because the default in C++ is that class members are mutable (e.g. a member with no annotations is mutable), it could be the case that a method does not modify state and yet the developer omitted a `const` annotation from it. We therefore wanted to know whether C++ developers `const`-annotated all of the methods that they could have annotated.

To our surprise, many of the all-mutating classes that we found did indeed have all member functions potentially modifying their containing class; developers were generally not just also under-labelling their code.

Our tool, Immutability Check, performs inference for a restricted subset of `const`-able methods. We found that it could propose improvements to the existing immutability annotations for a significant fraction of method implementations. We discuss the effectiveness of our inference tool, along with some cases where it is not sophisticated enough to make suggestions.

We explore two lines of questioning in this chapter. First, we investigate how developers use C++ `const` in practice, as witnessed through Application Programming Interface declarations. Second, we investigate questions related to `const` inference and the limitations of a simple inference approach.

Use of `const`. In our first set of research questions, we study how developers use `const` when declaring the members of a class. To understand aspects of how developers use `const`, we explored two questions:

(RQ4.1) how often are classes all-`const` (i.e. declared to be completely immutable)?

(RQ4.2) how often are classes free of `const` methods (all-mutating)?

Analysis and Evaluation. Our tool suggests `const` annotations in simple cases. We therefore further investigate whether our tool can successfully infer `const` annotations.

(RQ4.3) does our tool successfully identify enough new `const` annotations to be useful?

The contributions of this chapter are:

- We formulated a set of questions about the usage of immutability in C++ codebases: do developers write immutable classes? all-mutating classes? Why?
- We answered these questions for 6 moderate-to-large C++ case studies and identified trends in `const` usage that hold across our set of case studies. To our knowledge, we are the first to empirically investigate the use of `const` in actual codebases.
- We developed and implemented a novel static analysis that identifies methods that could easily be labelled `const` and applied it to our set of case studies, finding a significant number of such methods.

Our work contributes observations of actual developer behaviour across a variety of real-world codebases, and can thus inform developers about best practices today. Furthermore, programming language designers often add features to languages based on intuition and experience. (Tunnell Wilson et al. discuss shortcomings of alternate approaches to language design in [88].) In the future, we hope that our work can guide language designers when considering features to add to languages.

4.1 Motivation

Our research questions focus on three archetypal uses/non-uses of `const`: immutable classes, all-mutating classes, and easily `const`-able methods. An immutable class has no user-visible mutable state. An all-mutating class is never invariant under method calls. Finally, an easily `const`-able method is one that can quickly be seen to not change internal state.

Immutable class. The most straightforward example of an immutable class is an immutable tuple implementation. Consider, for instance, this `Point` class:

```
class Point {
  int x, y;
public:
  Point(int x, int y) : x(x), y(y) {}
  int getX() const { return x; }
  int getY() const { return y; }
};
```

Once a `Point` object is constructed, it cannot be changed.

All-mutating class. On the other hand, an all-mutating class contains no methods that do not modify state. Random number generators are typical examples:

```
class RandomNumberGenerator {
  int64_t state;
public:
  RandomNumberGenerator() { state = 17L; }
  int getNext() {
    state = state * 134775813 + 1;
    return state;
  }
};
```

Easily `const`-able methods. Finally, we are interested in methods that can easily be seen to be `const`-able. For instance, if method `Point::getX()` above was not `const`-annotated, our analysis would flag it as a potentially `const`-able method.

The core of our analysis tracks which expressions may contain a copy of a field value. We can conclude that there are no writes to a field in `getX` and that this method returns a copy of the field. Since it cannot mutate, and returns a field copy, this method ought to be `const`.

4.2 Technique

We designed an Immutability Check tool to answer our research questions, building on the LLVM compiler infrastructure [47] and the Clang frontend. Figure 4.1 shows where the Immutability Check tool fits in: it records compiler invocation commands executed during a build and re-runs the front-end stages to collect information on `const` usage, storing the results to a web-accessible database.

To improve its signal-to-noise ratio, Immutability Check ignores a number of classes and methods. Immutability Check ignores classes with no source code, abstract classes, and classes with no `public` members. It also ignores some more classes based on inheritance—classes which inherit from a standard library class within the `std` namespace; classes which

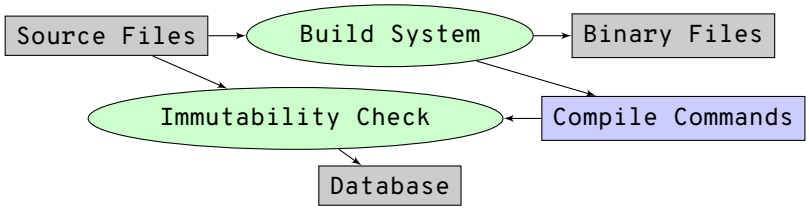


Figure 4.1: Our Immutability Check tool intercepts compile commands and stores results in a database.

inherit (transitively) from a class with no source; and classes whose inheritance hierarchy includes a template parameter. Within the set of included classes, we analyze all non-`static` public methods except for constructors/destructors, conversion operators, and `operator=`.

The C++ standard [72] allows developers to `const`-qualify methods and types. A class `C` effectively publishes two interfaces: a non-`const` (aka mutable) reference to `C` allows access to almost all of `C`'s members, while a `const` reference only allows access to `const`-qualified members. However, the `const` interface is not necessarily a subset of the mutable interface. Figure 4.2 shows how. In the Figure, class `C` has two `foo()` methods with the same signature, differing only by `const` qualifier. The two `foo()` methods are thus said to be `const`-overloaded. A `const`-qualified reference to `C` would expose the `const`-qualified `foo()` method, while a mutable reference to `C` would expose the non-`const`-qualified `foo()` method. Historically, this `const` overloading exists as a way to match the `const`-ness of a return value based off the callee. The canonical reason for this is to fix the "`strchr`" problem in C, so the `const`-ness of the input variable can propagate to the return value.

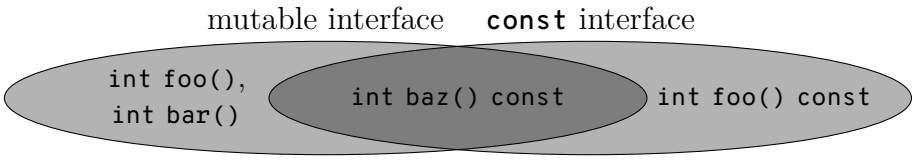
Immutability Check stores its results in a web-accessible database. This database holds the public members of every class, along with additional metadata. For methods, we store whether or not they are `const`-qualified along with the results of our analysis. For fields, we store whether or not they are `mutable`, and whether or not the outermost type is `const`-qualified.

Explicit `const` We call a field *explicitly* `const` if the field's outermost type is `const`-qualified. In that case, the field contains an object that is at least shallow immutable. A shallow immutable object has the property that the fields in the class itself do not change, but data pointed to by its fields may change. For instance, if a field of the object is a pointer, the pointer itself cannot change, but the data pointed to may. On the other hand, a transitively `const` type may not be mutated, and neither may anything obtained by dereferencing it.

To answer RQ4.1, we look for classes that only have `const`-qualified methods and explicitly `const` fields. We label such classes are considered "Immutable".

To answer RQ4.2, we look for classes without `const`-qualified methods and without fields. Such classes have all methods mutable and no accessors. A class may be considered "All-mutating" either if the developers neglected to use `const`, or if all methods could potentially change the class's state. To distinguish these cases, we investigated method implementations. We found that a nontrivial fraction of "All-mutating" classes indeed had

```
class C {
 public:
  int foo() { return 1; }
  int foo() const { return 2; }
  int bar() { return 3; }
  int baz() const { return 4; }
};
```




```
C c1;                              const C c2;
c1.foo(); // returns 1             c2.foo(); // returns 2
c1.bar(); // returns 3             c2.bar(); // not allowed
c1.baz(); // returns 4             c2.baz(); // returns 4
```

Figure 4.2: `const`-overloading implies that the mutable interface of a class is not a superset of the `const` interface.

all methods potentially mutate object state, which was somewhat surprising to us.

Static analysis. Our static analysis, implemented using Clang, is similar to an available expressions analysis. Our analysis computes whether or not a variable or expression must be a literal, or may be a field. The analysis is intraprocedural, flow sensitive, and path insensitive. Our join operation for literals is intersection, and union for fields. We make two unsound assumptions to lower the number of false positives in our analysis. The first assumption is that overloaded comparsions (e.g. `operator<`) never mutate. We do not expect this assumption to be violated in sane codebases, and it allows us to find trivial comparsion methods for custom classes. The second assumption is that global variables never point to objects' fields or contain values copied from fields.

Queries. We use the results of the static analysis to answer two queries about methods: whether or not a method mutates, and what the method returns. There are two options for whether a method mutates: 1) does not mutate, and 2) maybe mutates. There are three options for what a method returns: 1) noop (void or a literal), 2) field, and 3) other. For the mutation property, a method maybe mutates if either: 1) there is an assignment to an expression that may be a field, 2) there is a call to a member fuction, or 3) there is a call to a function where an argument may be a field. For the return property we join all return expressions.

Table 4.1: We chose 7 open-source codebases as case studies, ranging from 13 000 to over 3 million lines of code.

		kLOC	Classes	
LLVM	4.0.0	≈ 3,200	10521	compiler infrastructure
OpenCV	3.2.0	1,167	2221	computer vision library
Protobuf	3.3.1	625	407	compiler for data serializer/unserializer
fish	2.5.0	112	129	modern shell
mosh	1.2.6	14	86	mobile shell
ninja	1.7.2	13	36	build system
libsequence	1.8.7	18	33	library for evolutionary genetics

Transitivity To further reduce the number of false positives, we consider the return type of the method. Consider a class with disjoint mutable and `const` interfaces. Typically, these interfaces would be disjoint due to pointer accessor functions. The mutable method returns a pointer without a `const` qualifier to the object. Unsurprisingly, the `const` method returns a pointer to a `const`-qualified object. We do not want to report the mutable method as easily `const`-able, even though it just returns a field. To be sure we don't report the mutable version as a false positive, we ensure the return type is transitively `const`. That is, the return type must not allow mutation through indirection (e.g. `const int * getP();` does not allow its return value to be dereferenced and then mutated). Transitively `const` is the strictest definition of `const`, and reduces the number of potential false positives. Note that, since C++ is pass by value, we do not consider the `const` qualifier on the outermost type. Our definition of transitivity is incomplete when the transitivity depends on the internals of the class (for example, if the class contains a bare pointer); our results do include some false positives that are not properly filtered by transitivity.

Easily `const`-able methods Combining answers from our queries, and transitivity, we can determine whether or not a method is easily `const`-able. An easily `const`-able method: 1) is not already `const`-qualified; 1) does not mutate (according to our query); and 2) if our query says that the method returns a field, the return type must be transitively `const`.

4.3 Results

We evaluated 7 open source projects: fish, libsequence, LLVM, Mosh, Ninja, OpenCV, and Protobuf. We chose these projects because they are popular, well maintained, and span a variety of domains, including I/O-focussed applications as well as both graph-manipulating and array-manipulating codes. Table 4.1 lists characteristics of our chosen projects.

We present two major tables for each case study. The first table shows the overall distribution of all classes with a public interface. The second table shows the number of methods which are easily `const`-able.

Reading the class tables Following Figure 4.3, we divide classes into 6 categories. The first two categories, "Immutable" and "Query", only have `const` methods. "Immutable"

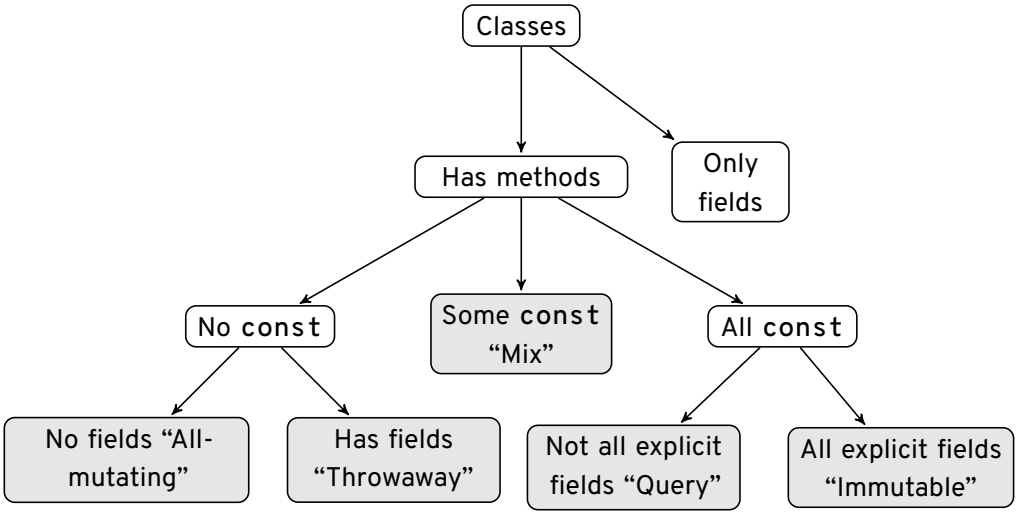


Figure 4.3: Immutability Check divides classes with methods into 5 main categories, depending on which members they contain.

has every public field explicitly declared `const`. On the other hand, "Query" has fields which may be modified by member functions when accessed through a non-`const` reference. "Mix" classes have both `const` and non-`const` methods. The last two categories, "Throwaway" and "All-mutating", both only have non-`const` methods. The "Throwaway" classes have fields, but these fields can only be read. Such classes therefore have accessors that could trivially be made `const`. We call these classes throwaway since there are no const qualifiers at all, and also public fields. Typically such classes are a quick and dirty structure. "All-mutating" classes, by contrast, do not have fields; we believe that all methods in such classes should mutate. Finally, we show the number of classes with only fields (no methods) for completeness.

For answering RQ4.1 and RQ4.2 we only consider non-trivial classes. A non-trivial class has more than 3 methods. To estimate the number of immutable and all-mutating classes, we take the appropriate categories (from the developer labelling) and manually sample them to find the true number of these classes. For larger projects where we cannot reasonably manually check all classes, we randomly sample 40 classes (20 of each category).

Reading the easily `const`-able methods tables The first row of the table shows the number of methods in the project, which covers 100% of the total methods in the project (indicated by % total). The next row shows how many easily `const`-able methods we find using our "trust" analysis across all methods, ignoring developer labelling. Next, we present how many easily `const`-able methods we find using our base analysis, without trust. In this row we show relative percentages to the total number of methods, and to the number of methods found with the trust analysis. For instance, if the relative % to trusted is 50%, then the "trust" analysis found twice as many easily `const`-able methods as the base analysis. The middle 3 rows are similar to the previous 3 rows, except all the methods included have a `const` label provided by the developers. This allows us to determine the

percentage of more simple `const` methods. The final 3 rows are similar to the previous sets of 3, except that they describe the methods where developers did not include a `const` label. The easily `const`-able entries for these methods without `const` show potentially missing `const` labels.

4.3.1 LLVM

Compilers extensively manipulate structured intermediate representations, and must maintain invariants to preserve the meaning of the code being compiled. We expect that recording design intent with respect to immutability would be key to successfully developing LLVM.

The LLVM compiler infrastructure [47] is our largest case study. We ran our tool on version 4.0.0, which has approximately 3,246,000 lines of code. (We did not include unit tests in our analysis, but we included auto-generated code.) Table 4.2 shows our overall results. Because LLVM has thousands of classes, we randomly sampled 20 all-immutable and all-mutating classes for manual analysis.

Table 4.2: Most LLVM classes contain a mix of `const` and non-`const` methods, but about a quarter of its classes are immutable or all-mutating.

Classes	10 521	100%
Has methods	8 395	79.8%
Immutable	1 135	10.8%
Query	474	4.5%
Mix	4 762	45.3%
Throwaway	568	5.4%
All-mutating	1 456	13.8%
Only fields	2 126	20.2%

Table 4.3: All sampled LLVM classes that developers declared "Immutable" are in fact immutable; only 8/20 "all-mutating" classes are all-mutating. Additionally, 4 "all-mutating" classes were in fact immutable.

Classes	Total	Non-trivial	Sampled	Actual
Immutable	1 135	582	20	20
All-mutating	1 456	546	20	8

[RQ4.1a, RQ4.2a] Table 4.3 shows that at a class level, we expect that about $663 = 582 + 15\% \times 546$ of the classes in LLVM will be non-trivial (more than 3 methods) immutable classes: our sampling shows that about 100% of the `const`-annotated immutable classes are immutable, while about 15% of the non-trivial classes annotated as all-mutating were immutable. This accounts for about 6% of the total number of LLVM classes. On the other hand, we expect that 218 classes will be non-trivial all-mutating, or about 2% of the total number of classes. A plurality of classes, and a majority of classes with methods, contained a mix of `const` and mutable methods. **[RQ4.1b, RQ4.2b]** We found that classes that

Table 4.4: LLVM methods, immutability declarations, and easily `const`-able methods. LLVM contains a plurality of `const`-labelled methods, and about half of LLVM methods are easily `const`-able. % developer-labelled indicates the percentage of easily `const`-able methods that carry a `const` label. % trusted indicates the percentage of methods that are easily `const`-able without the trust assumption compared to with it.

	Total	% total	% developer-labelled	% trusted
All methods	213 968	100%		
Easily `const`-able (trust)	106 631	49.8%		
Easily `const`-able (base)	70 703	33.0%		66.3%
`const` methods	129 582	60.6%		
Easily `const`-able (trust)	86 060	40.2%	66.4%	
Easily `const`-able (base)	51 482	24.1%	39.7%	59.8%
Non-`const` methods	84 386	39.4%		
Easily `const`-able (trust)	20 571	9.6%	24.4%	
Easily `const`-able (base)	19 221	9.0%	22.8%	93.4%

developers labelled as immutable were indeed immutable, while classes that lacked `const` declarations could have used them on some of their member functions more often than not. [**RQ4.3**] Table 4.4 shows that, at a method-level granularity, developers declared about 60% of methods to be `const`, while we found that about 50% of methods could be easily seen to be immutable by the "trust" variant of our analysis (and 33% by the non-"trust" variant). Furthermore, about 66% of the developer-labelled `const` methods were easily-`const`-able (with "trust"), while only 24% of non-`const` methods were easily-`const`-able. Interestingly, the trust assumption has little effect on non-`const` methods; the analysis identifies 93% as many `const`-able methods with the trust assumption as without it.

Discussion. Many of the immutable classes that we manually inspected in LLVM were either checker or code generator classes. Code generator classes change program state (or emit side effects), but do not change the state of the receiver object. We also encountered some immutable code generator classes in our inspection of all-mutating classes; these classes simply did not have any `const`-labelled methods.

4.3.2 OpenCV

To explore numeric codes, we studied the OpenCV computer vision library. The parts of OpenCV that carry out regular calculations on large arrays are more amenable to parallelization than codes that process graphs. As expected, we found that OpenCV mutated many of its arrays. However, we found some usage of `const` on function parameters. We ran our tool on version 3.2.0 with 1,167,551 lines of code. Table 4.5 presents our overall results.

Table 4.5: OpenCV contains relatively many immutable classes and fewer all-mutating classes.

Classes	2 221	100%
Has methods	1 607	72.4%
Immutable	332	14.9%
Query	246	11.1%
Mix	712	32.1%
Throwaway	129	5.8%
All-mutating	188	8.5%
Only fields	614	27.6%

Table 4.6: Fewer than a third of OpenCV's immutable and all-mutating classes are non-trivial. Manual inspection showed that almost all sampled classes declared immutable are immutable, while a quarter of all-mutating classes are all-mutating.

Classes	Total	Non-trivial	Sampled	Actual
Immutable	332	45	20	18
All-mutating	188	62	20	5

[**RQ4.1a, RQ4.2a**] For OpenCV, Table 4.6 shows that we expect 90% of 45 non-trivial classes to be immutable, or about 41, which accounts for 1.8% of its classes. (Many of OpenCV's classes are trivial.) None of the sampled all-mutating classes were immutable. Hence, we expect 25% of 62 non-trivial classes to be all-mutating, or about 16, which accounts for 0.7% of its classes. [**RQ4.1b, RQ4.2b**] Again, classes that were labelled as immutable often were immutable, while all-mutating classes often could have had some methods `const`-labelled (identified by the analysis).

[**RQ4.3**] Table 4.7 shows that, once again, developers `const`-labelled a 54% majority of methods. For OpenCV, about 13% more methods were easily `const`-able, which makes up about 68% of methods overall. The "trust" variant of the easily `const`-able analysis finds about twice as many methods to be easily `const`-able among methods that are already `const`-labelled, but does not contribute much for non-`const`-labelled methods.

Discussion. OpenCV contains many implementations of mathematical functions. It appears that the convention in OpenCV is to implement these functions within classes (similar to the Strategy design pattern). An alternate system design could have manipulated these functions using function pointers. In any case, OpenCV's function classes fit the definition of an immutable object. However, these function classes appear to often be trivial (fewer than 4 methods).

Table 4.7: OpenCV methods, immutability declarations, and easily `const`-able methods. A plurality of OpenCV methods are `const`-labelled, and 38% are easily `const`-able. % developer-labelled indicates the percentage of easily `const`-able methods that carry a `const` label. % trusted indicates the percentage of methods that are easily `const`-able without the trust assumption compared to with it.

	Total	% total	% developer	% trusted
All methods	16 864	100%		
Easily `const`-able (trust)	6 503	38.6%		
Easily `const`-able (base)	3 797	22.5%		58.4%
`const` methods	9 139	54.2%		
Easily `const`-able (trust)	5 449	32.3%	59.6%	
Easily `const`-able (base)	2 769	16.4%	30.3%	50.8%
Non-`const` methods	7 725	45.8%		
Easily `const`-able (trust)	1 054	6.2%	13.6%	
Easily `const`-able (base)	1 028	6.1%	13.3%	97.5%

4.3.3 Protobuf

Protobuf (protocol buffers) serializes structured data. We analyzed the protocol buffer compiler, which generates code (in a number of languages) to serialize and deserialize to/from specified data formats. This compiler also happens to contain generated protocol buffer code. We ran our tool on version 3.3.1 of Protobuf, which had 625,820 lines of code. Table 4.8 shows our overall results.

Table 4.8: Protobuf has the highest proportion of immutable classes among our studied codebases.

Classes	407	100%
Has methods	352	86.5%
Immutable	107	26.3%
Query	4	1.0%
Mix	133	32.7%
Throwaway	2	0.5%
All-mutating	106	26.0%
Only fields	55	13.5%

[**RQ4.1a, RQ4.2a**] Based on the results in Table 4.9, in Protobuf's case, we estimate that about 82 of 407 (20%) classes are immutable—that is 95% of 67 non-trivial `const`-labelled classes, plus 50% of 36 non-trivial classes with no `const` labels. Furthermore, we expect about 8 (2%) non-trivial all-mutating classes, or 20% of 36. [**RQ4.1b, RQ4.2b**] Protobuf shows similar proportions as OpenCV for all-`const`-labelled classes being immutable and non-`const`-labelled classes being all-mutating. However, Protobuf has a significantly higher proportion of immutable classes that are free of `const` labels than our other case studies (50%).

Table 4.9: About half of Protobuf's classes are non-trivial. Manual inspection showed that almost all sampled classes declared immutable are immutable, while 20% of all-mutating classes are all-mutating. Additionally, 10 "all-mutating" classes were in fact immutable.

Classes	Total	Non-trivial	Sampled	Actual
Immutable	107	67	20	19
All-mutating	106	36	20	4

Table 4.10: Protobuf methods, immutability declarations, and easily `const`-able methods. Almost two-thirds of Protobuf methods are `const`-labelled, and 33% are easily `const`-able. % developer-labelled indicates the percentage of easily `const`-able methods labelled `const`. % trusted indicates methods that are easily `const`-able without the trust assumption compared to with it.

	Total	% total	% developer	% trusted
All methods	5 893	100%		
Easily `const`-able (trust)	1 959	33.2%		
Easily `const`-able (base)	782	13.3%		39.9%
`const` methods	3 753	63.7%		
Easily `const`-able (trust)	1 812	30.7%	48.3%	
Easily `const`-able (base)	640	10.9%	17.1%	35.3%
Non-`const` methods	2 140	36.3%		
Easily `const`-able (trust)	147	2.5%	6.9%	
Easily `const`-able (base)	142	2.4%	6.6%	96.6%

[**RQ4.3**] Table 4.10 shows that almost two-thirds of Protobuf's methods are immutable. However, most of these immutable methods were already declared as `const`; our easily `const`-able analysis (both with and without "trust") does not contribute much. The Protobuf developers used `const` for complex methods: when looking at methods that the developers labelled `const`, the easily `const`-able analysis only found about half, which is less than for other benchmarks. The trust assumption almost triples the number of methods found to be easily `const`-able.

Discussion. Protobuf, like LLVM, contains many classes that generate code. When we manually inspected them, we verified that they were immutable. Some of the immutable classes had no `const` labels and hence had originally been identified as all-mutating classes.

4.3.4 fish shell

The fish shell (`fishshell.com`) is a modern command line shell. Shells are particularly concerned with file-based I/O. We ran our experiments on version 2.5.0, which has 112,951 lines of code. Table 4.11 shows our overall findings.

Since fish and the subsequent benchmarks have no more than 20 classes in each category, we exhaustively examined each of the immutable and all-mutating classes.

Table 4.11: Many fish classes contain a mix of `const` and non-`const` methods, but a surprisingly high proportion of fish classes are all-mutating.

Classes	129	100%
Has methods	89	69.0%
Immutable	8	6.2%
Query	23	17.8%
Mix	27	20.9%
Throwaway	11	8.5%
All-mutating	20	15.5%
Only fields	40	31.0%

Table 4.12: fish has no non-trivial immutable classes and only 6 non-trivial classes with no `const` annotations, of which only 2 are all-mutating.

Classes	Total	Non-trivial	Actual
Immutable	8	0	0
All-mutating	20	6	2

Table 4.13: fish methods, immutability declarations, and easily `const`-able methods. 43% of fish's methods are `const`-labelled, and 41% are easily `const`-able. % developer-labelled indicates the percentage of easily `const`-able methods labelled `const`. % trusted indicates methods that are easily `const`-able without the trust assumption compared to with it.

	Total	% total	% developer	% trusted
All methods	341	100%		
Easily `const`-able (trust)	138	40.5%		
Easily `const`-able (base)	99	29.0%		71.7%
`const` methods	145	42.5%		
Easily `const`-able (trust)	112	32.8%	77.2%	
Easily `const`-able (base)	73	21.4%	50.3%	65.2%
Non-`const` methods	196	57.5%		
Easily `const`-able (trust)	26	7.6%	13.3%	
Easily `const`-able (base)	26	7.6%	13.3%	100.0%

[**RQ4.1a, RQ4.2a**] As seen in Table 4.12, we expect fish to have 0 (and hence 0%) non-trivial immutable classes. It is possible that some of the classes that have some, but not all, `const`-labelled methods might actually be fully immutable; however, we believe that this is unlikely—if developers have `const`-labelled some of the immutable methods, they would be likely to have labelled all of them. Of the 6 non-trivial classes that lack `const`-annotated methods, 2 (1.5%) of them are non-trivial all-mutating. [**RQ4.1b, RQ4.2b**] Since we estimate that fish contains 0 immutable classes, we believe that developers did not leave out any `const` annotations on the immutable classes. About a third of classes without any `const` annotations could have used some.

[**RQ4.3**] Table 4.13 shows that, even though fish has no immutable classes, it still has immutable methods. Developers `const`-qualified 43% of methods, and we found another 8% to be easily `const`-able with "trust", which makes up about half of the methods for this application. Many (77%) `const`-qualified methods were easily `const`-able with "trust", while only half could be `const`-able without it. Once again, the trust assumption has little effect on methods without `const` annotations.

4.3.5 libsequence

The libsequence project is a library for evolutionary genetics. This library is primarily a collection of mathematical functions. Mathematical functions are a use case where we would expect mutation to be kept at a minimum. We ran our tool on version 1.8.7 of libsequence which contains approximately 18,000 lines of code. We present our overall results in Table 4.14.

Table 4.14: Approximately half of libsequence's classes are immutable, while the other half contain a mix of `const` and non-`const` classes. No libsequence classes are all-mutating.

Classes	33	100%
Has methods	29	87.9%
Immutable	16	48.5%
Query	1	3.0%
Mix	11	33.3%
Throwaway	1	3.0%
All-mutating	0	0.0%
Only fields	4	12.1%

Table 4.15: All libsequence non-trivial classes that developers declare "Immutable" are in fact immutable. libsequence contains 0 all-mutating classes.

Classes	Total	Non-trivial	Actual
Immutable	16	8	8
All-mutating	0	0	0

[**RQ4.1a, RQ4.2a**] For libsequence, Table 4.15 shows 8 non-trivial immutable classes, as manually verified, accounting for 25% of all classes. However, libsequence has no all-mutating classes: every class in libsequence has at least one `const`-labelled method.
[**RQ4.1b, RQ4.2b**] Our manual inspection, which was exhaustive in this case, did not find any missing `const` annotations on the all-mutating classes nor any inappropriate `const` annotations on the immutable classes.
[**RQ4.3**] As seen in Table 4.16, the libsequence developers labelled 91% of libsequence's methods as `const`, which is the highest in our set of programs. Our easily `const`-able analysis only found 39% of this 91%, even with the "trust" assumption. Without the

Table 4.16: All libsequence methods, immutability declarations, and easily `const`-able methods. 91% of libsequence's methods are `const`-labelled, yet only 40% are easily `const`-able. % developer-labelled indicates the percentage of easily `const`-able methods labelled `const`. % trusted indicates methods that are easily `const`-able without the trust assumption compared to with it.

	Total	% total	% developer	% trusted
All methods	211	100%		
Easily `const`-able (trust)	84	39.8%		
Easily `const`-able (base)	71	33.6%		84.5%
`const` methods	192	91.0%		
Easily `const`-able (trust)	82	38.9%	42.7%	
Easily `const`-able (base)	70	33.2%	36.5%	85.4%
Non-`const` methods	19	9.0%		
Easily `const`-able (trust)	2	0.9%	10.5%	
Easily `const`-able (base)	1	0.5%	5.3%	50.0%

"trust" assumption, our easily `const`-able analysis found 85% as many immutable methods. Additionally, we only found another 2 methods (1% of the overall method count) to be easily `const`-able that were not already `const`-labelled.

Discussion. Like OpenCV, libsequence contains many mathematical functions. The developers have also already `const`-annotated these functions. Our easily `const`-able analysis does not find many immutable functions for libsequence, so it would have been difficult to understand its immutability structure without the developer annotations. Note that both variants of the analysis can only check just over a third of `const` methods.

4.3.6 Mosh

Mosh is a utility for maintaining terminal connections over low-quality (e.g. cellular data) networks. It sends and receives data over the network, encrypting and decrypting the data as necessary. We ran our tool on version 1.2.6 which contains 14 415 lines of code. Table 4.17 shows our overall findings.

[**RQ4.1, RQ4.2**] Table 4.18 shows that we expect Mosh to have 0 non-trivial immutable classes and 0 non-trivial all-mutating classes. Again, it is possible but unlikely that Mosh actually has immutable classes with some but not all methods `const`-annotated.

[**RQ4.3**] Mosh has a typical proportion of immutable methods, as seen in Table 4.19. Here, we see that developers `const`-labelled 54% of the methods and that an additional 5% are easily `const`-able, for a total of 59%. The easily `const`-able analysis identifies 45% of methods as immutable, which is 83% of the number of `const`-labelled methods. "Trust" helps identify about 8% more methods as being immutable.

Discussion. Because Mosh is a smaller project, developers might not believe there is an advantage to creating immutable classes (or maybe the domain). The developers, however,

Table 4.17: A (relatively) smaller proportion of Mosh classes are immutable or all-mutating compared to other benchmarks.

Classes	74	100%
Has methods	68	91.9%
Immutable	5	6.8%
Query	3	4.1%
Mix	52	70.3%
Throwaway	2	2.7%
All-mutating	6	8.1%
Only fields	6	8.1%

Table 4.18: Mosh has 0 non-trivial immutable classes and 0 non-trivial all-mutating classes.

Classes	Total	Non-trivial	Actual
Immutable	5	0	0
All-mutating	6	0	0

Table 4.19: Mosh methods, immutability declarations, and easily `const`-able methods. 54% of Mosh methods are `const`-labelled, and 45% are easily `const`-able. % developer-labelled indicates the percentage of easily `const`-able methods with a `const` label. % trusted indicates the percentage of methods that are easily `const`-able without the trust assumption compared to with it.

	Total	% total	% developer	% trusted
All methods	416	100%		
Easily `const`-able (trust)	185	44.5%		
Easily `const`-able (base)	153	36.8%		82.7%
`const` methods	225	54.1%		
Easily `const`-able (trust)	164	39.4%	72.9%	
Easily `const`-able (base)	135	32.5%	60.0%	82.3%
Non-`const` methods	191	45.9%		
Easily `const`-able (trust)	21	5.0%	11.0%	
Easily `const`-able (base)	18	4.3%	9.4%	85.7%

did seem to take care to `const`-label methods.

4.3.7 Ninja

The Ninja build system creates a dependency graph and runs commands to rebuild targets when their dependencies change. Ninja outsources almost everything to other tools—notably the calculation of the dependency graph. It focuses on its core functionality—the processing of the dependency graph and the selection of the appropriate commands to execute. We ran our tool on version 1.7.2, which is 13,662 lines of code. Table 4.20 shows our

overall results.

Table 4.20: Ninja has many all-mutating classes and few immutable classes.

Classes	36	100%
Has methods	32	88.9%
Immutable	1	2.8%
Query	2	5.6%
Mix	17	47.2%
Throwaway	5	13.9%
All-mutating	7	19.4%
Only fields	4	11.1%

Table 4.21: Ninja has 0 non-trivial immutable classes and 0 non-trivial all-mutating classes.

Classes	Total	Non-trivial	Actual
Immutable	1	0	0
All-mutating	7	2	0

Table 4.22: Ninja methods, immutability declarations, and easily `const`-able methods. 22% of Ninja methods are `const`-labelled, and 26% are easily `const`-able. % developer-labelled indicates the percentage of easily `const`-able methods with a `const` label. % trusted indicates the percentage of methods that are easily `const`-able without the trust assumption compared to with it.

	Total	% total	% developer	% trusted
All methods	165	100%		
Easily `const`-able (trust)	43	26.1%		
Easily `const`-able (base)	36	21.8%		83.7%
`const` methods	36	21.8%		
Easily `const`-able (trust)	23	13.9%	63.9%	
Easily `const`-able (base)	18	10.9%	50.0%	78.3%
Non-`const` methods	129	78.2%		
Easily `const`-able (trust)	20	12.1%	15.5%	
Easily `const`-able (base)	18	10.9%	14.0%	90.0%

[**RQ4.1, RQ4.2**] Table 4.21 shows that we also expect Ninja to have 0 non-trivial immutable classes and 0 non-trivial all-mutating classes.
[**RQ4.3**] Ninja has a lower-than-usual proportion of immutable methods, as seen in Table 4.19. Here, we see that developers `const`-labelled only 22% of the methods and that an additional 12% are easily `const`-able, for a total of 34%. The easily `const`-able analysis identifies 26% of methods as immutable, which is more than the number of methods that the developers labelled `const`. "Trust" only helps identify a few more methods as being immutable.

Discussion. This lower-than-usual proporation of immutable methods may be due to the problem domain. Ninja, as a build system, typically mutates its dependency tree as a result of running commands. These commands cause mutations in system state which Ninja needs to reflect.

4.3.8 Summary

Table 4.23 shows our overall false positive rates. We identified the false positives by randomly sampling and manually inspecting 20 methods each of `const` and non-`const` with both our "trust" and base analysis, up to a maximum of 560 methods across the 7 projects. If there were fewer than 20 methods in a category, we manually inspected all methods. Note that the 50% false positive rate for libsequence was due to only having 2 classes in that category. Overall, we found our false positive rate for `const`-labelled methods to be very low for the "trust" analysis, and zero for the base analysis. For non-`const` methods, our false positive rate is between 5-6% for either analysis, which is useful for developers.

Table 4.23: Our false positive rates across all projects is low (see note for libsequence). Our analysis has few false positives for `const` methods and an acceptable number for non-`const` methods.

	`const` methods % false positives (trust)	`const` methods % false positives (base)	non-`const` methods % false positives (trust)	non-`const` methods % false positives (base)
LLVM	0	0	5	0
OpenCV	0	0	5	10
Protobuf	5	0	20	10
fish shell	0	0	5	10
libsequence	0	0	50	0
Mosh	0	0	0	0
Ninja	0	0	0	0
Overall	0.7	0	6.6	5.1

We summarize our findings with respect to the research questions. For RQ4.1 and RQ4.2, we verified the classes by manually inspecting every class in the smaller projects and 60 randomly-selected classes across the larger projects (20 per project).

[**RQ4.1a.**] Do C++ developers reveal a preference for writing all-immutable classes?
[**Finding 4.1a.**] Across the projects we surveyed, developers only declare a median of 2% non-trivial all-immutable classes (with a range between 0 and 25%).

[**RQ4.1b.**] How effective are they at labelling classes where all methods are immutable?
[**Finding 1b.**] Across the immutable classes we manually inspected, we found that 96% of them were properly `const`-annotated.

Table 4.24: Summary of results. (RQ4.1, 2) A median of 2% of the classes that developers write are immutable, and the same percentage is all-mutating. (RQ4.3a) Developers write a far greater number of immutable methods than immutable classes, with a median of 54% across our case studies. (RQ4.3b) Compared to the number of immutable methods that they declare to be `const`, developers could declare an additional 13% (median) methods to be `const`.

	% non-trivial immutable classes	% non-trivial all-mutating classes	% declared `const` methods	% undeclared easily `const`-able methods
LLVM	6	2	70	24
OpenCV	2	1	60	14
Protobuf	20	2	65	7
fish shell	0	2	49	13
libsequence	25	0	92	10
Mosh	0	0	59	11
Ninja	0	0	22	16

[**RQ4.2a.**] Do C++ developers reveal a preference for writing all-mutating classes?
[**Finding 4.2a.**] Across the projects we surveyed, developers declare between 0 and 2% non-trivial all-mutating classes (median 1%).

[**RQ4.2b.**] How effective are they at labelling classes where all methods mutate the receiver object?
[**Finding 4.2b.**] We found that 28% of non-trivial all-mutating classes we manually inspected were indeed all-mutating.

We then shifted our focus to counting the number of immutable methods that should exist in a codebase. First, we used the developer-provided `const` label as an approximation of that number, but we also added an estimated number of un-annotated methods. Second, to measure effectiveness, we noted that it is impossible to add `const` annotations to a class if all of its methods are already `const`. Therefore, when answering RQ4.2b, we count the remaining classes, which are classes that could have a `const` qualifier added.

[**RQ4.3a.**] Do C++ developers write methods (rather than classes) that should be labelled as immutable?
[**Finding 3a.**] We found that developers declare a median of 54% of methods to be immutable using the `const` qualifier. Furthermore, we estimate that at least 60% of methods (median) are in fact immutable.

[**RQ4.3b.**] How effective are C++ developers at labelling immutable methods?
[**Finding 4.3b.**] Of the potential methods that could have a `const` method, developers fail to label a median of 13%.

We found that the "trust" variant of our easily `const`-able analysis was far more effective

when analyzing `const` methods than when analyzing non-`const` methods. Recall that the "trust" variant relies on `const` annotations to exist on callee methods, and such annotations are far more likely to exist when the method under analysis is already `const`.

Implications to language design. We formulated our research questions to guide language design around immutability. RQ4.1 addresses how much developers could benefit from support for declaring immutable objects. Our results show that non-trivial immutable objects were rare across our case studies. Therefore, we would advocate that developers' needs are not served by simply adding support for immutable objects. RQ4.2 addresses whether or not developers would notice a lack of immutability annotations. Consider that, if a significant number of classes contain methods that all mutate state, then developers would not benefit from immutability labels for those classes. We found very few (but, to our surprise, more than zero) non-trivial classes that only contained mutating methods. Switching our focus from classes to methods, we found that the ability to label methods as immutable is useful. RQ4.3 shows that developers label a majority of their methods as immutable, which exceeded our expectations. We also found that even more methods could be labelled as immutable, suggesting that tools to enable developers to add missing `const` annotations to their already-immutable code would be helpful.

Further Analysis Insights. This work shows that developers do fail to label some `const` methods which could be easily labelled. However, it also shows that a simpler analysis (even one that trusts `const` callees) is not sufficient to check most immutable methods. The trust variant of the analysis did show that an analysis that checks immutability must be interprocedural. Trust typically doubled the number of immutable methods. We believe that the results from this chapter and Chapter 3 illustrate the need for an interprocedural analysis that can check for abstract immutability.

Chapter 5

Abstract Immutability Analysis

The ability to mutate state is the distinguishing feature of imperative programming languages. It is ubiquitous in today's popular object-oriented programming languages like C++ and Java. Mutation's disadvantages are well known: in particular, unexpected side effects can send developers on lengthy debugging missions. Modern languages therefore include some support for controlling mutability—for instance, `const` in C++ and `final` in Java, among others. In principle, protecting parts of the program state from change can improve the quality of software abstractions and reduce developers' cognitive load. However, support for immutability in today's mainstream languages is quite limited, as noted by Coblenz et al. [14].

C++ includes the `const` keyword, which provides some support for immutability. According to the C++ Foundation's FAQ [24], C++ `const` is intended to mean "logically const". We specifically interpret "logically const" to mean abstract immutability. That is, while the bits physically representing an object may change even across a `const` operation, the abstract (or logical) state of the object is not allowed to change. Quoting from the FAQ:

> This is going to get inane, but let's be precise about whether a method changes the object's logical state. If you are outside the class—you are a normal user, every experiment you could perform (every method or sequence of methods you call) would have the same results (same return values, same exceptions or lack of exceptions) irrespective of whether you first called that lookup method. If the lookup function changed any future behavior of any future method (not just making it faster but changed the outcome, changed the return value, changed the exception), then the lookup method changed the object's logical state—it is a mutator. But if the lookup method changed nothing other than perhaps making some things faster, then it is an inspector.
>
> — C++ Foundation FAQ on `const`

Current state-of-the-art techniques, both for C++ and for Java immutability systems, protect fields from mutation in `const` methods. However, usability considerations dictate that these techniques also allow developers to exempt fields declared as `mutable` from protection under `const`. Existing systems therefore allow arbitrary changes to mutable fields in `const` methods, and the values of such fields can be exposed to the user. Given

the existence of `mutable`, it is currently largely the responsibility of the developer to guarantee "logical const", which undermines the safety that could potentially be provided by proper support for immutabilty.

Our goal is to enable developers to maintain classes that implement abstract immutability. In particular, we intend our tool to be part of a continuous integration workflow: the tool would verify that class implementations continue to be abstractly immutable, or else point out methods that may mutate abstract state yet are labelled `const`. We believe that support for verified abstract immutability can help with program understanding and evolution by preventing unintended side effects. The key technical insight behind our analysis is that abstract immutability implementations like caches may write to a class's internal state, but that such writes are permissible as long as they precede reads of the same fields. Note that there are other cases of abstractly immutable class we would not be able to identifiy, however a sound version of our analysis verifies a subset of all abstractly immutable classes. Our analysis does not require any developer annotations.

Our contributions include:

- a class-based static immutability analysis which identifies abstractly immutable class implementations, even in the presence of controlled writes;
- an implementation of our analysis in the LLVM framework; and
- an evaluation of our approach on sizeable C++ programs.

5.1 Motivating Example

We next demonstrate our technique on a pair of motivating examples. Our analysis identifies classes that are abstractly immutable, even if they contain `const` methods that mutate fields (i.e. are not concretely immutable). When a class is not logically `const`, our analysis identifies the writes that potentially break that class's `const`-ness.

Our goal. Our immutability analysis establishes that: **if an object is only accessed through its `const` interface, then no writes will modify any exposed fields.** An exposed field affects a value returned to a caller through an explicit value flow (see [50] for a discussion of explicit vs implicit flows and their impact on static analysis). A positive answer from our analysis therefore implies that calling any `const` method on an object will not change subsequent return values from any other `const` method on that object, satisfying the C++ Foundation's recommendation for `const` usage.

Our analysis establishes class-level properties by analyzing all of the methods of a particular class, including any methods that it may inherit from superclasses. Although our analysis visits transitive callees of the methods in the class under analysis, making conservative assumptions when needed, it is not a whole-program analysis: it does not visit methods unreachable from the class under analysis. Hence, to obtain a result about abstract immutability of a subclass, the developer must explicitly analyze the subclass.

We emphasize that C++'s `const` qualifier makes no guarantee about abstract immutability: if `mutable` fields exist, it is the responsibility of the developer to ensure abstract

immutability. In particular, `const` methods may violate abstract immutability by returning values derived from `mutable` fields despite the presence of mutations to those fields. Our analysis identifies such violating methods.

We acknowledge that non-`const` references to `A` objects may exist alongside `const` references to the same objects. In other words, in C++, a developer may modify an object with `const` references to it, by using a non-`const` reference to that object. Our analysis aims to show that if a developer only uses `const` references to an object, then the object's abstract state will not change. C++ programmers must use coding conventions to ensure that no unexpected non-`const` references are shared.

We continue with an implementation of a standard caching lookup. Recall that, in Chapter 3, we reported that 3 of the widely-used C++ case studies use `const` objects with caches. Listing 5.1 contains lookup method `getValue()`, which satisfies C++'s requirements for `const` methods: it calls only `const` methods and contains only writes to `mutable` fields. The `getValue()` method also calls an `unrelated()` function, which has a do-nothing implementation in class `A`; we return to this function later. For now, observe that `getValue()` is abstractly immutable when the receiver object is of type `A`, since it contains no writes that change fields visible to callers.

Listing 5.1: Typical (correct) caching implementation in `getValue()` method.

```
1  class A {
2  protected:
3    mutable int value;
4    mutable bool cached;
5
6    virtual void unrelated() const {}
7
8  public:
9    // C++ allows setCached to be const because cached is mutable,
10   // but setCached violates abstract immutability.
11   void setCached(bool c) { cached = c; }
12
13   int getValue() const {
14     unrelated();
15     if (!cached) {
16       value = 3;
17       cached = true;
18     }
19     return value;
20   }
21 };
```

The `getValue()` implementation in `A` is abstractly immutable as long as method `unrelated()` does not modify the `cached` and `value` fields which influence `getValue()`'s return value. Consider, then, Listing 5.2, which shows a subclass `B` for which `getValue()` is no longer abstractly immutable. `B`'s method may return either `3` or `42`, depending on the value of the `cached` field. Note that `B`'s `getValue()` has the same implementation as

`A.getValue()` and still satisfies the requirements for C++'s `const` qualifier. Note also that `B.unrelated()`, on its own, is abstractly immutable—`B.unrelated()` does not contain any field reads.

Listing 5.2: An implementation for `unrelated()` that breaks `A`'s abstract immutability.

```
class B : A {
protected:
  virtual void unrelated() const {
    value = 42;
  }
};
```

Analysis of abstractly immutable class `A` As discussed above, our immutability analysis proceeds on a per-class basis. We will first describe how our analysis handles class `A`. This class contains two `const` methods, `getValue()` and `unrelated()`.

Upon entry to `getValue()` (i.e. before Line 14) in Listing 5.1, our analysis constructs the abstract state in Figure 5.1. This graph starts with local variable `%this`, which is (in this case) LLVM's representation for `this`. The local variable contains a pointer node. The pointer node, in turn, points to a struct node (LLVM terminology). This struct node represents an instance of an `A` object and itself contains unknown-valued int nodes for fields `cached` (on the left) and `value`.

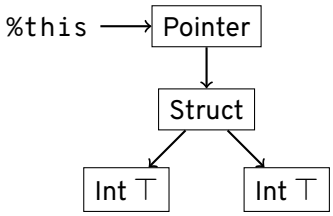


Figure 5.1: Heap abstraction upon entry to `getValue()` showing that `this` points to an object (structure node) of type `A` with unknown field values. Boxes indicate sequential or structure nodes, while edges indicate pointee or struct edges. Unboxed node `%this` indicates variable binding.

The analysis then encounters the call to `unrelated()`. For this per-class analysis, there is only one possible callee for `unrelated()`: the empty implementation in `A`. The analysis visits the empty body and returns with the same abstraction. (Our analysis continues visiting callees until the flow of control hits a recursive call or leaves the class being analyzed. It makes worst-case assumptions when it skips a callee.)

Next, the analysis processes the `if` statement. It uses the `if`-condition to refine the abstraction and produces Figure 5.2a for the true branch and Figure 5.3 for the false branch. Abstractly executing the body of the true branch yields Figure 5.2b. Notice how the true branch contains known values for both fields, while the false branch knows only that `cached` is true but not the contents of `value`. The control-flow merge gives the

abstraction in Figure 5.4, which has `cached` known true and `value` shown as $3 \cup \top$ for legibility (actually $\top$).

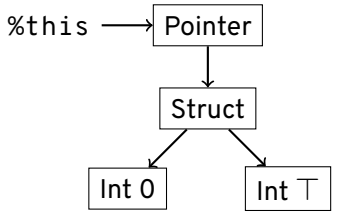
A key part of our immutability analysis consists of ensuring that exposed fields are not subsequently modified. `getValue()`'s return statement exposes the `this` object's `value` field, so we will report any modifications of it in future passes. (Figure 5.4 shows such nodes with grey backgrounds.)

More generally, at return statements, the analysis uses its abstraction to check whether any edges from the node representing the return value reach any nodes representing values from the `this` object. If so, then it marks all reachable nodes as being read. Any writes to these fields in subsequent calls will cause an error.

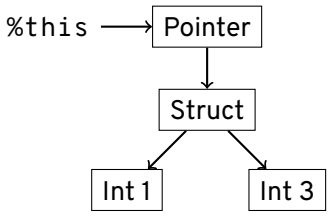
We iterate the per-method analysis on all public `const` methods until we reach a fixed-point. Between iterations, we only keep the abstraction for state reachable from `this` through directed edges.

In this case, a second analysis of `getValue()` checks that calls to `getValue()` will never store to fields that were read by any public `const` method of `A`. This second analysis iteration changes nothing in the abstraction, so the analysis has reached its fixed point and terminates.

For class `A`, our analysis concludes that there are no writes modifying exposed field `value`, and thus that the public `const` interface of `A` is abstractly immutable.



(a) Abstraction before line 16.



(b) Abstraction after line 17.

Figure 5.2: Heap abstraction, `cached == false` branch (condition true).

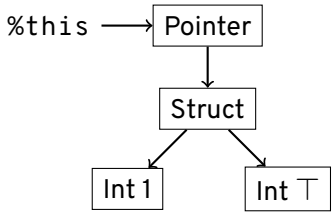


Figure 5.3: Heap abstraction, `cached == true` branch (condition false).

Analysis of non-abstractly immutable class `B` We next consider class `B`, whose `unrelated()` implementation modifies field `value`. The analysis once again starts with method `getValue()`. This time, the call to `unrelated()` resolves to `B`'s implementation. Figure 5.5 shows the heap abstraction upon exit from `unrelated()`[1].

[1] Our actual abstraction is somewhat more complicated because of subclassing and virtual dispatch, but the differences do not affect this analysis result.

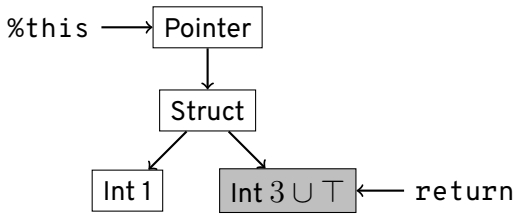


Figure 5.4: Heap abstraction after return (line 19).

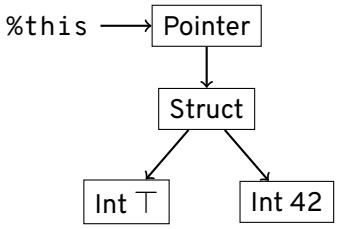


Figure 5.5: Heap abstraction upon return from `B.unrelated()`.

After the analysis processes all of the statements in the method, it obtains the abstract state in Figure 5.6. The first analysis of `getValue()` once again identifies the int $3 \cup 42$ node as the value being returned, and marks it as a node that is read from.

The analysis then revisits `getValue()`, working towards a fixed point. This time, when it analyzes `unrelated()`, it observes the write to the `value` field, which has been marked as being returned (grey background), and thus flags an immutability violation error at that write. The developer can investigate the immutability violation and correct it.

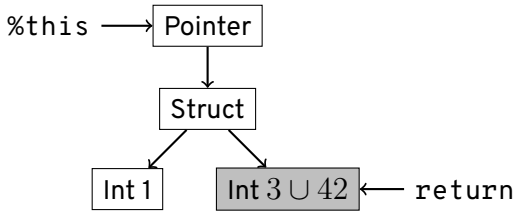


Figure 5.6: Heap abstraction in `B.getValue()` after merge (line 18).

5.2 Technique

Section 5.1 presented our analysis via examples. In this section, we more formally describe our analysis, starting with the abstraction, which tracks information about which values have been exposed and about aliasing in the heap. This abstraction enables the analysis to deduce, at a per-class level, that no exposed value is modified by a `const` method. Note that this analysis does not detect all cases of abstract immutability. It would capture caching properties, but not others such as a move-to-front optimization. We continue by explaining our transfer functions and finally present the merge operation. Because we implemented our system on top of LLVM [47], we use many terms from its compiler infrastructure, which we will explain as needed.

5.2.1 Formalization

The most general definition of abstract immutability is that a caller cannot observe any changes to the object, across any sequence of method calls. Our formalization describes a dynamic property which captures a subset of this most general definition; it prohibits writes to a field after that field has been exposed (e.g. returned to a caller). Our static analysis flags any prohibited writes.

We describe our approach using a minimal language. The syntax we use is similar to Featherweight Java [40]. Unlike Featherweight Java, our syntax includes field writes. The important parts of the syntax are field reads/writes, method calls, and return statements. We present the syntax below:

Class	:=	`class` ClassName {*Field* Method**}
Field	:=	*Type* fieldId
Method	:=	*Type* methodId (*Arg**) { *Stmt** } (implicit `this` argument)
Arg	:=	*Type* argId
Type	:=	ClassName | `int` | *Type**
Stmt	:=	`return` *Expr*
		Expr
		o = `new` ClassName (object creation)
		$o.f$ = *Expr* (field write)
		x = *Expr* (local variable write)
Expr	:=	$o.f$ (field read)
		x (local variable read)
		$o.m$(*Expr**) (method call)

This grammar represents a minimal version of classes, similar to those in C++ and Java. A class can have many methods and fields. Without loss of generality, we assume fields behave as `private` fields in C++ and Java, because `public` fields break encapsulation. To model `public` fields we could replace them with accessor methods. A method consists of a list of arguments (with `this` always being present) and a sequence of statements. The important statements for the analysis are field writes and returns. The return statement only returns primitive values.

Concrete semantics. Let V be the set of root variables, F be the set of field names, O be the set of typed heap objects, and T be the set of types, $T = \{\text{Struct},\ \text{Pointer},\ \text{Primitive}\}$. Objects can hold primitive values.

$$\textit{TypeOf}\colon \begin{cases} O \to T \\ o \mapsto t \end{cases}$$

$$\textit{Element}\colon \begin{cases} O \times F \to O \\ (o, f) \mapsto e \end{cases} \quad \text{where } \textit{TypeOf}(o) \in \{\text{Pointer}, \text{Struct}\}$$

$$\textit{Value}\colon \begin{cases} O \to \mathbb{Z} \\ o \mapsto x \end{cases} \quad \text{where } \textit{TypeOf}(o) = \text{Primitive}$$

$$\textit{Location}\colon \begin{cases} V \to O \\ v \mapsto o \end{cases}$$

These functions represent the state of the heap. We track pointees of pointers and fields of structures through the *Element* function. We model a pointer as a structure with a single field value representing the pointee. The *Value* function contains the value of primitive objects. The final function, *Location*, maps variables to objects in the heap. Let `this` be a distinguished variable, *TypeOf*(*Location*(`this`)) = Pointer, representing the receiver object of method calls.

Definition 5.2.1 *A heap, h, consists of valuations for TypeOf, Element, Value, Element along with sets V, F, and O.*

Definition 5.2.2 *Let $Reachable(h, v)$ be a function that returns the set of objects reachable from variable v.*

$Reachable(h, v) = \bigcup_{i \in \{x \mid 1 \leq x \leq |O|\}} S_i$

$S_1 = \{Location(v)\}$

$S_i = \bigcup_{o \in S_{i-1}} Next(o)$

where $Next(o) = \{Element(o, f) \mid \forall f \in F\}$

Definition 5.2.3 *Let predicate $ReachableFromThis(h, o)$ be true when object o is reachable through Element, rooted from the distinguished* `this`*. The set of objects that satisfies ReachableFromThis is $Reachable(h,$* `this`*$)$.*

Definition 5.2.4 *A trace, tr, is a sequence of pairs, $\langle h, pc \rangle$, where pc is the program counter and activation records. The successor relation in a valid trace, $tr_i \rightarrow tr_{i+1}$, respects the operational semantics of our language [40].*

Definition 5.2.5 *A developer-provided set of* `const` *methods, M_C, satisfies $M_C \subseteq M$ for class C.*

We represent a C++ class's set of `const` methods using M_C, and our analysis checks abstract immutability for the methods in M_C. Proposition 1 states that if our (sound) analysis does not produce a warning, then there can be no writes to any exposed fields in any future calls to any method in M_C.

We use the next definition to differentiate actions that are encapsulated within an object from actions that are externally-visible. In particular, this definition identifies calls and returns that cross the object's encapsulation boundary.

Definition 5.2.6 *Given a trace tr, a boundary for object o with method m (where $m \in M_C$) consists of an entry,* `o.m()`*, in tr and the corresponding exit from m,* `return v`*. The receiver object o, upon entering the boundary entry, is bound to* `this`*.*

Definition 5.2.7 *Let predicate $PreviouslyReturned(o, tr)$ be true when an object o was returned to the outside world previously on that trace. Specifically, at a boundary exit (*`return` *v), on the trace tr, we add all objects in $Reachable(v)$ to $PreviouslyReturned(o, tr)$.*

The reason that Definition 5.2.7 only adds objects at boundaries is that within a boundary, any further method calls on the same `this` are considered internal, and do not return values to the original caller (unless object o is subsequently returned at the boundary exit).

Definition 5.2.8 *Let predicate IsExposed(o, tr) be true when an object o satisfies predicates ReachableFromThis(h, o) and PreviouslyReturned(o, tr).*

Definition 5.2.9 *A valid heap for pc is any heap reachable from an empty heap by executing statements according to the operational semantics beginning at the entry point of the program until reaching pc. An empty heap contains empty valuations for all of its mappings.*

Definition 5.2.10 *Let predicate IsAbstractlyImmutable(M) be true when a set of methods M on a distinguished* `this` *object o can not write to fields of o where IsExposed(o, tr) for any trace tr.*

Proposition 1 *(Soundness) If the analysis analyzes a set of methods, M_C, starting with a maximal abstract heap, and returns no warnings, then: given a trace tr with final program point pc, a valid heap for pc, and a distinguished object* `this` *in the heap, for all objects o such that IsExposed(o, tr) is true, then there is no write to o in either tr or any extensions of tr. In this case the predicate IsAbstractlyImmutable is true for M_C.*

5.2.2 Analysis Operation

Recall that our analysis considers each class in isolation to determine whether all of its `const` member functions are logically `const`. Our analysis computes heap abstractions for member functions to track exposed nodes, i.e. nodes with their "read" property set in the abstraction. A method is logically `const` if it is free of writes to exposed nodes. After analyzing a method, the analysis discards everything not reachable from `this`, and continues analyzing other `const` member functions until it reaches a fixed point. This corresponds to analyzing a method at the boundary; it takes into account all possible sequence of method calls. If all `const` methods are logically `const`, then the class is logically `const`—calling a `const` method never causes a change in the object's visible state. Since we have reached a fixed point no other method calls (corresponding to M_C in the formalization) may write to a "read" (or exposed) value. This corresponds to no extension of a dynamic trace being able to write to an exposed value in the formalization.

5.2.3 Abstraction

We use a novel class-based approach for our analysis. The analysis considers one class at a time, attempting to show that the class's public methods' implementations respect abstract immutability. We represent the (intra-class but inter-procedural) analysis state with a graph $G_p = (N, E)$ for each program point p. Nodes in these graphs represent abstract heap objects or constant values, while edges represent pointer and aliasing relationships between nodes. Our graphs include directed edges for pointer relationships and undirected edges for aliasing relationships.

Nodes. At its core, our analysis tracks which values may be exposed to callers. Since our analysis is built on LLVM, we map each LLVM value to a node. Nodes represent heap objects and integer values. We focus on three kinds of LLVM nodes: pointer nodes N_{Pointer},

structure nodes N_{Struct}, and integer nodes N_{Int}. Each abstract pointer and structure node represents one or more concrete heap objects. Integer nodes represent constant integer values. LLVM pointer nodes represent pointers while structure nodes represent object instances. (LLVM also has floating point and function nodes, but they are not relevant to our analysis: we do not consider floating point; and we handle function calls when analyzing `call` and `return` instructions.) Nodes that are no longer reachable from LLVM variables may be omitted from a graph.

Because our analysis seeks to show that `this` remains abstractly unchanged across `const` methods, it tracks a boolean "this" flag for each node. A node n's "this" flag is true when n represents an object currently reachable (along pointers and field references) from the `this` object as defined at a given program point. If the "this" boolean is true for a node, this corresponds with the predicate $\mathit{ReachableFromThis}(h, o)$ being true from Definition 5.2.3.

For each node with true "this" flag, our analysis also tracks a boolean "read" property. Our analysis verifies that nodes with "read" set to true are never written to after being marked as "read". In our illustrations, we use a grey background to denote "read" being true. The initial value for "read" is false. If the "read" boolean is true for a node, this corresponds with the predicate $\mathit{IsExposed}(o, tr)$ being true from Definition 5.2.7.

Our analysis tracks nullness for each pointer node using a standard 4-element lattice (impossible/definitely-null/definitely-not-null/anything). LLVM's struct nodes are like pointer nodes for structures (including classes); our abstraction uses edges to represent a map, keyed on field names, from struct nodes to field contents.

We also track possible values for integer nodes. At these nodes, we explicitly use $\top$ to represent "any value", $\bot$ to represent "impossible", and ranges otherwise.

Edges. Edges in our graph represent pointer and aliasing relationships between nodes. Our analysis maintains four types of directed edges between nodes: pointee edges, struct edges, substruct edges, and this edges. To determine all objects that we reachable, we follow pointee and struct edges. It also maintains a set of undirected edges between nodes: weak edges. Finally, our analysis maintains a mapping between LLVM variables (which are not nodes) and graph nodes.

Directed edges are as follows:

- "Pointee" directed edges $e \in E_{\text{Pointer}}$ link pointers to pointees. Every pointer node has at most one pointee edge out of it—in our abstraction, a pointer may only point to one pointee. A pointer node without an outgoing pointer may point to any object of appropriate type, as tracked by our memory alias sets (described below). Directed edges, including pointee edges, respect types; for instance, a pointee edge could go from a pointer `int32 *` node to an `int32` int node, but not vice-versa.

- "Struct" directed edges $e \in E_{\text{Struct}}$ link struct nodes with their field contents. Index labels indicate which field.

- "Substruct" edges $e \in E_{\text{SubStruct}}$ handle subtyping relations between objects. If class `B` is a subclass of `A`, and if we are analyzing in the context that `this` is a `B`, then any "this" node representing an instance of an `A` object will have a substruct edge to the

`B` subclass. This maintains object state, so that virtual methods on `B` that require the subtype `B` from an `A` can find one.

- "This" edges $e \in E_{\text{This}}$ between nodes represent explicit flow from a node with "this" flag true to any other node in the graph (regardless of type). For instance, after statement `t = x.f+1`, where `x` has "this" flag true (and hence so does `f`), then our analysis will create a "this" edge from the "t" to the int node for "f". The purpose of these edges is to conservatively approximate the nodes that are in *ReachableFromThis*.

Undirected edges are as follows:

- "Weak" edges $e \in E_{\text{Weak}}$ indicate that two values may be aliased; a strong update of one of the values associated with the edge requires a weak update of the other. We denote weak edges with dotted lines.

  As Figure 5.7 illustrates, however, the relationship is not between the connected nodes, but rather between the nodes that point to the connected nodes. That is, if pointer node A points to int node int_A and pointer node B points to int node int_B, then a weak edge between the int nodes indicates that pointer nodes A and B may point to the same object. A strong update to A's pointee must trigger a weak update to B's pointee, and vice-versa. Weak edges arise due to control-flow merges.

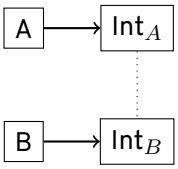


Figure 5.7: Weak edge (dotted) indicates nodes A, B may point to same object.

LLVM variable mapping edges E_{Var} associate each LLVM variable (e.g. %0) with a node.

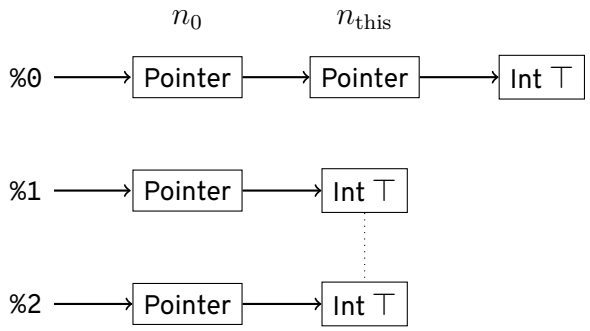
All together,

$$E = E_{\text{Pointer}} \cup E_{\text{Struct}} \cup E_{\text{SubStruct}} \cup E_{\text{This}} \cup E_{\text{Weak}} \cup E_{\text{Var}}.$$

Initial state. Our analysis starts analyzing a method in the initial state where the LLVM variable associated with `this` (typically `%0`) points to a pointer node n_0. Node n_0 then has a pointee edge to a struct node n_{this} representing the `this` object. The struct node's int fields are initialized to fresh int nodes containing $\top$, while other fields get pointer nodes with no aliases and outgoing pointers. Note that this does not represent a maximal heap, which we use in the formal definition. If we assumed the `this` object could contain aliases, it would model a maximal heap, however practically this makes the analysis unusable for any objects with multiple pointer fields. In other ways our analysis is more conservative. Our analysis considers initial object state that may not be possible in executions of the program. For instance, there may be an invariant set in an object's constructor. Method parameters are also initialized to fresh pointer nodes n_p; however, the objects that the n_p

point to may alias existing objects in the heap. All together, the analysis assumes that there is a `this` object with unaliased fields, and nothing about its contents.

A method with two `int *` parameters on a class with one `int` field would have this initial abstraction:



The weak edge indicates that `%1` and `%2` may alias. We assume `%1` and `%2` don't alias fields of parameters.

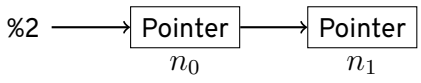
5.2.4 Running Example

Figure 5.8 presents the LLVM control-flow graph for code similar to `getValue()` from Section 5.1. We will use this CFG to illustrate our transfer functions.

Section 5.2.5 presents key transfer functions for our analysis. We explain the functions below.

`ID = alloca TYPE.` The LLVM `alloca` instruction allocates a fresh memory location. Our analysis thus creates a fresh pointer node and a fresh node containing the default value for the corresponding type, along with edges from the LLVM variable to the new pointer node and from the new pointer node to the new instance of the default value.

Instruction `%2 = alloca %class.A*` adds fragment:



`ID = getelementptr PTR(, INDEX)+.` This instruction creates a pointer node which provides direct access to a pointee or a field node.

The example contains statement `%4 = getelementptr %3, 0, 1`, which creates a pointer that follows the pointer edge followed by the struct edge for `cached`. For this graph only, we've labelled the E_{Pointer} edges with indices; everywhere else, we omit indices for clarity. Our analysis creates fresh pointer node $n_{\%4}$ bound to variable `%4`, along with a pointee edge from $n_{\%4}$ to n_v (which it arrives at by following indices $0, 1$ starting from $n_{\%3}$), yielding the following graph after analyzing `getelementptr`:

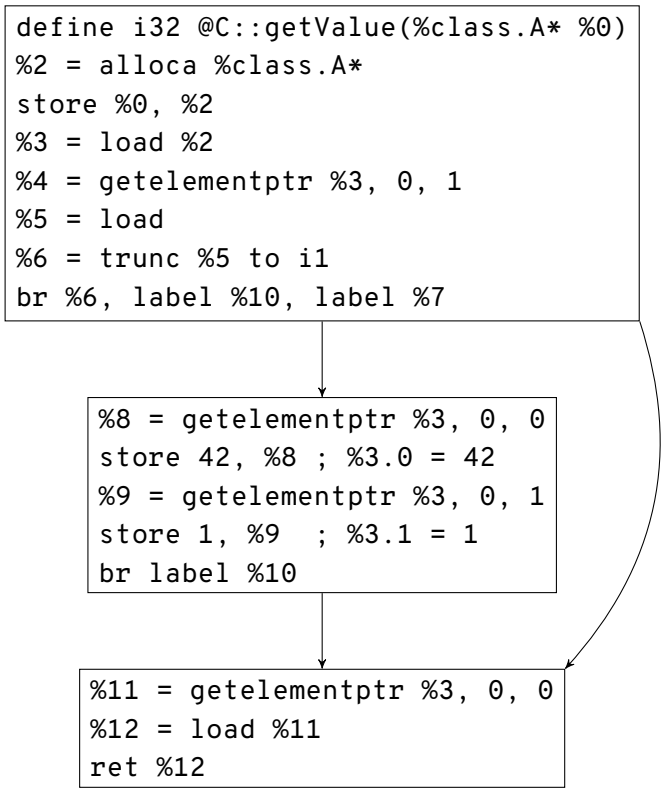
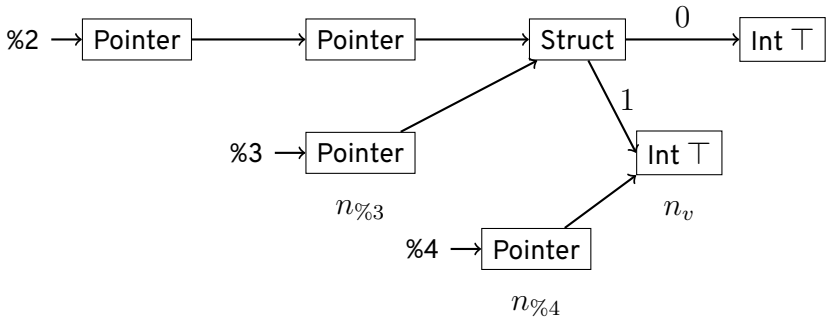


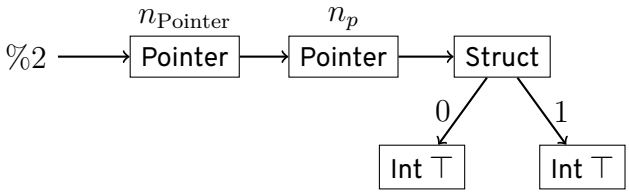
Figure 5.8: LLVM control-flow graph inspired by `A` in the motivating example.



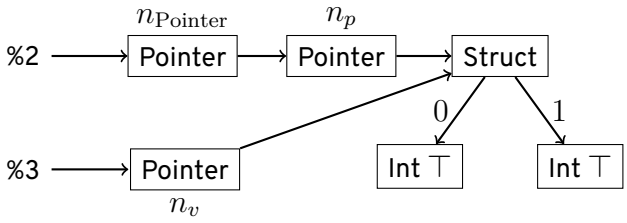
`ID = load PTR.` The LLVM `load` instruction loads a value from memory at address `PTR` and puts the value into `ID`, i.e. `ID = *PTR`. Our analysis creates a fresh pointer node which points to the pointee (`*PTR`). The new node also gets weak and "this" edges patterned after the node that it was copied from.

In the example, `%3 = load %2` would have before and after states:

Before:



After:



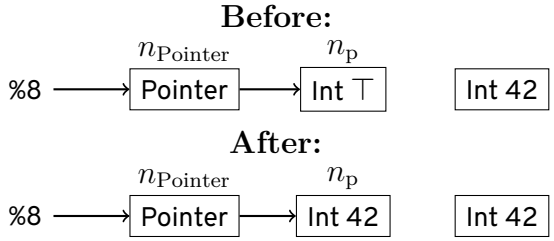
This shows how `%3` gets a fresh pointer node that points to n_p's pointee.

`store VAL, PTR.` The LLVM `store` instruction stores a value to the location referenced by a pointer (i.e. `*PTR = VAL`). Our analysis therefore updates the contents of the location pointed-to by `PTR` and relevant "this" edges.

Let `PTR` map to pointer node $n_{\text{pointer}} \in N_{\text{Pointer}}$ (i.e. $(\texttt{PTR}, n_{\text{Pointer}}) \in E_{\text{Var}}$), and let `VAL` map to node n_v. Node n_{pointer} may have one pointee edge; let the target of that edge (i.e. `*PTR`) be node n_p. (If no such edge exists, our analysis makes the conservative assumption that n_{pointer} points to a fresh pointee object, weakly aliased with all other objects of appropriate type; if n_{pointer} has "this" flag true, our analysis assumes the new pointee object is unique.)

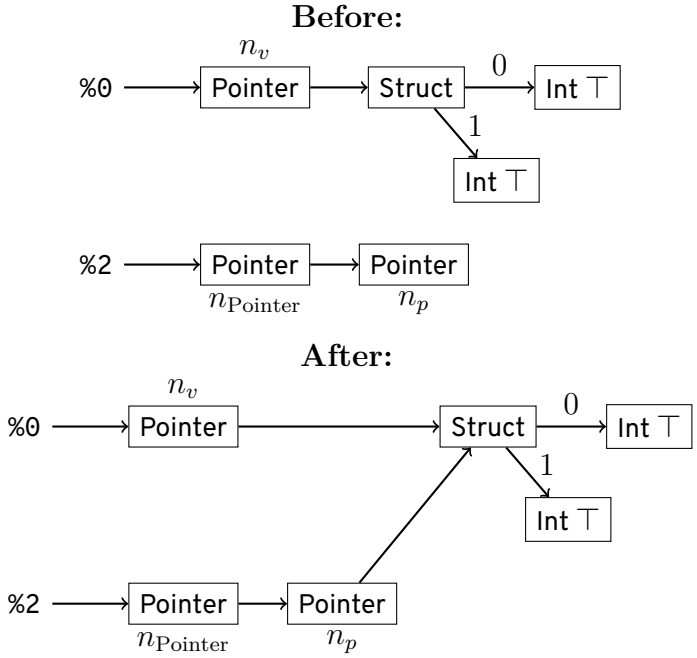
The analysis updates "this" edges and replaces the contents of n_p with those of n_v. It removes all "this" edges involving n_p, reflecting the fact that n_p's value has changed. If n_v is a "this" node, it adds a "this" edge (n_p, n_v). Finally, for all "this" edges (n_v, n'_v), the analysis creates "this" edges (n_p, n'_v).

In the example, `store 42, %8` would have these before and after states:



If n_p is a pointer node (i.e. `PTR` is a pointer to a pointer), the analysis will change pointee edges out of n_p to reflect the change. It removes any edges $(n_p, *)$ from E_{Pointer} and adds a new edge (n_p, n_v).

In our example, we have statement `store %0, %2`, which results in the addition of an edge from pointer node n_p to struct node n_v.



5.2.5 Transfer Functions

We next present the transfer functions underlying our analysis. Our transfer functions determine whether **`const`** methods are abstractly immutable with respect to the distinguished `this` object, by tracking which parts of `this` have been made visible to callers and verifying that these parts do not subsequently get modified. The transfer functions update heap abstractions across LLVM statements.

freshNode(`T`) means a new, non-null node representing an object of type `T`.

freshWeak(`T`) means freshNode(`T`) with weak edges to all objects of appropriate type.

`ID = alloca TYPE.` Let:

$$n_0 := \text{freshNode}(\texttt{TYPE}\ *) \qquad n_1 := \text{freshNode}(\texttt{TYPE}).$$

Then:

$$\begin{aligned} N' &= N \cup \{n_0, n_1\}; \\ E'_{\text{Var}} &= E_{\text{Var}} \setminus (\texttt{ID}, *) \cup (\texttt{ID}, n_0);\ \text{and}, \\ E'_{\text{Pointer}} &= E_{\text{Pointer}} \cup \{(n_0, n_1)\}. \end{aligned}$$

`ID = getelementptr PTR(, INDEX)*.` Let:

$$(\texttt{PTR}, n_0) \in E_{\text{Var}} \qquad (n_i, n_{i+1}) \in E_{\text{Pointer}} \cup E_{\text{Struct}}$$

$$\begin{aligned} \text{where } [n_0, \ldots, n_k] \ &\text{are} \ \text{the specified indices;} \\ n_{\text{pointer}} \ &:= \ \text{freshNode}(\text{typeof}(\texttt{ID})). \end{aligned}$$

Then:

$$\begin{aligned} N' &= N \cup \{n_{\text{Pointer}}\}; \\ E'_{\text{Var}} &= E_{\text{Var}} \setminus \{(\texttt{ID}, *)\} \cup \{(\texttt{ID}, n_{\text{pointer}})\};\ \text{and} \\ E'_{\text{Pointer}} &= E_{\text{Pointer}} \cup \{(n_{\text{pointer}}, n_k)\}. \end{aligned}$$

`ID = load PTR.` Let:

$$(\texttt{PTR}, n_{\text{pointer}}) \in E_{\text{Var}}$$

$$\begin{aligned} (n_{\text{pointer}}, n_p) \ &\in \ E_{\text{pointer}} \text{ if such an } n_p \text{ exists;} \\ &\quad\ \text{else } n_p \text{ is freshWeak(typeof(}\texttt{ID}\text{))}. \\ \text{if } n_p \text{ pointer: } (n_p, n'_p) \ &\in \ E_{\text{pointer}} \text{ such an } n'_p \text{ exists;} \\ &\quad\ \text{else } n'_p \text{ is freshWeak(typeof(}\texttt{ID}\text{))}. \end{aligned}$$

Then:

$$\begin{aligned} N' &= N \cup \{n_v = \text{freshNode}(\text{typeof}(\texttt{ID}))\} \\ E'_{\text{Var}} &= E_{\text{Var}} \setminus \{(\texttt{ID}, *)\} \cup \{(\texttt{ID}, n_v)\} \\ E'_{\text{this}} &= E_{\text{this}} \ \cup\ \{(n_v, n_p)\} \text{ if } n_p \text{ has ``this'' true;} \\ &\qquad\ \cup\ \{(n_v, n_t) \text{ for all } n_t \text{ such that } (n_p, n_t) \in E_{\text{this}}; \end{aligned}$$

$$\begin{aligned} &\text{if } n_p \text{ int node:} && \text{copy contents of } n_p \text{ to } n_v; \\ &\text{if } n_p \text{ pointer node:} \quad E'_{\text{Pointer}} &&= E_{\text{Pointer}} \cup \{(n_v, n'_p)\}. \end{aligned}$$

`store VAL, PTR.` Let:

$$\begin{aligned}(\texttt{VAL}, n_v), (\texttt{PTR}, n_{\text{pointer}}) &\in E_{\text{Var}} \\ n_p &= \text{if it exists,} \\ &\quad \text{the unique node } \{n_p \mid (n_{\text{pointer}}, n_p) \in E_{\text{Pointer}}\}; \\ &\quad \text{else, freshWeak(typeof(}\texttt{VAL}\text{))}.\end{aligned}$$

Our analysis updates edges:

$$\begin{aligned}E'_{\text{this}} &= E_{\text{this}} \setminus \{(n_p, *)\} \\ &\quad \cup \{(n_p, n_v)\} \text{ if } n_v \text{ has "this" true;} \\ &\quad \cup \{(n_p, n'_v) \text{ for all } n'_v \text{ such that } (n_v, n'_v) \in E_{\text{this}};\end{aligned}$$

$$\begin{array}{lrll}\text{if } n_p \text{ Int node:} & & & \text{copy contents of } n_v \text{ to } n_p; \\ \text{if } n_p \text{ Pointer node:} & E'_{\text{Pointer}} & = & E_{\text{Pointer}} \setminus \{(n_p, *)\} \cup \{(n_p, n_v)\}; \\ & & \text{plus} & \text{weak updates for } n_p \text{ (see text).}\end{array}$$

5.2.6 Memory Alias Sets

Our analysis uses a type-based approach to conservatively estimate worst-case aliasing relationships between objects. That is, for each type, the analysis tracks all potentially pointed-to nodes of that type. The analysis uses this information to perform worst-case updates across functions that it does not analyze. In addition, for structs (including classes), our analysis encodes the C++ implementation guarantee that different fields of the same struct instance do not alias.

Our per-statement graph abstraction therefore includes 4 maps. Specifically, we store: 1) known pointees, 2) unknown pointees, 3) known fields, and 4) unknown fields. Maps 1 and 2 are keyed by the type of the node. Maps 3 and 4 are keyed by the type of the node, the containing struct type, and the field. Let T be a type, S be a struct type, F be a field name, and N be a set of nodes. Formally, the resulting maps are as follows:

1. $\text{Mem}_{\text{Known}} : T \to N$,
2. $\text{Mem}_{\text{Unknown}} : T \to N$,
3. $\text{Mem}_{\text{KnownField}} : T \times S \times F \to N$, and
4. $\text{Mem}_{\text{UnknownField}} : T \times S \times F \to N$.

Note that any nodes created with proper allocation (such as `alloca`) are added to the known sets. If there is any pointee or field accesses with no information, we create a new node and add it to the corresponding unknown set. For instance, if there is an uninitialized pointer, we create a new node of the appropriate type, which may alias any other node of the same type. Maps 3 and 4 are for fields. If there is an uninitialized struct that has 2 fields of the same type, we do not want these fields to alias.

These memory alias sets add additional weak edges to our abstraction, as follows:

Unknown. Let:

$$n_0 \in \mathrm{Mem}_{\mathrm{Unknown}}(t_0).$$

Then:

$$(n_0, n_1) \in E_{\mathrm{Weak}}$$

where

$$\begin{aligned} n_1 \in \quad & \mathrm{Mem}_{\mathrm{Unknown}}(t_0), n_0 \neq n_1 \\ \cup \quad & \mathrm{Mem}_{\mathrm{Known}}(t_0) \\ \cup \quad & \mathrm{Mem}_{\mathrm{Unknown}}(t_0, *, *) \\ \cup \quad & \mathrm{Mem}_{\mathrm{KnownField}}(t_0, *, *). \end{aligned}$$

Intuitively, an unknown pointee may alias any other node of the same type (even if that node is part of a struct).

Unknown Fields. Let:

$$n_0 \in \mathrm{Mem}_{\mathrm{UnknownField}}(t_0, s_0, f_0).$$

Then:

$$(n_0, n_1) \in E_{\mathrm{Weak}}$$

where

$$\begin{aligned} n_1 \in \quad & \mathrm{Mem}_{\mathrm{UnknownField}}(t_0, s_0, f_0), n_0 \neq n_1 \\ \cup \quad & \mathrm{Mem}_{\mathrm{KnownField}}(t_0, s_0, f_0). \end{aligned}$$

Intuitively, an unknown field may alias any other matching field. Note that fields may still alias unknown pointees from the previous rule.

Other Transfer Functions

We include descriptions of our other transfer functions below. Because they are straightforward, we chose to omit them from Section 5.2.5.

Arithmetic operations. In addition to updating the ranges at int nodes, our analysis also updates "this" edges. In particular, at an operation `%z = ADD %x, %y`, let n_x, n_y, and n_z be the relevant nodes after E_{Var} lookups. If there is an edge $(n_x, n_t) \in E_{\mathrm{This}}$, then we propagate the "this" property by adding edge (n_z, n_t) to E_{This} as well. (Note that our analysis guarantees that n_t will have had its "this" flag set to true.)

`branch`. Our analysis performs predicated analysis at conditionals; for instance, at `br %6, label %10, label %7`, it propagates `%6` true at successor `%10` and `%6` false at successor `%7`, by adding relevant E_{Var} edges.

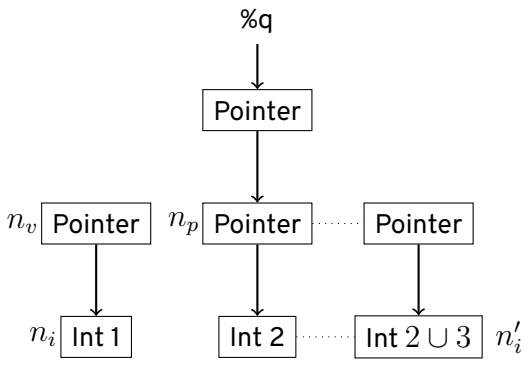
`call`. Our analysis analyzes the `call` LLVM instruction by leveraging the fact that it is a class-based analysis. Specifically, it continues across method boundaries if the callee remains in the class under analysis, and makes worst-case assumptions if the callee is outside the class. Because the analysis is focussed on a particular class, it assumes that "this" is an object of that exact class. This allows the analysis to devirtualize the method call.

Once the analysis has identified the callee, it maps the caller's actuals to the callee's formals and continues the analysis at the callee's entry point. After the call completes, the analysis removes any mappings specific to the callee.

`return`. At a return statement, the analysis propagates the program state back to the caller, binding the callee's return value to the appropriate variable in the caller.

At a top-level return statement, our analysis also marks state as escaping or being read. Recall that the analysis considers each public `const` method in turn, iterating on the set of methods until it reaches a fixed point. Top-level return statements are the statements which return from these public `const` methods. Figure 5.9 illustrates our algorithm for handling top-level returns. At a top-level `return %r`, the analysis sets the "read" flag for all "this"-flagged nodes reachable from `%r`. The analysis also reports, upon return, any immediate pointees of the formal method parameters that are "this"-flagged; such nodes may escape the method.

Weak Updates. Our analysis also performs weak updates as required by the weak edges in the graph. Consider a store `store v, %q` with the following initial state:



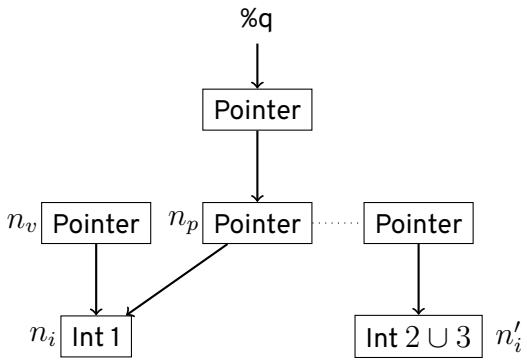
First, the analysis swings the n_p edge in E_{Pointer}, adding an edge to n_v's pointee:

```
1: define immediatePointees(n):
2: if n ∈ N_Pointer then
3:    S' := S ∪ {n_p} where (n, n_p) ∈ E_Pointer
4: else if n ∈ N_struct then
5:    S' := immediatePointees(n_f) where (n, n_f) ∈ E_struct
6: end if
7: return S'
```

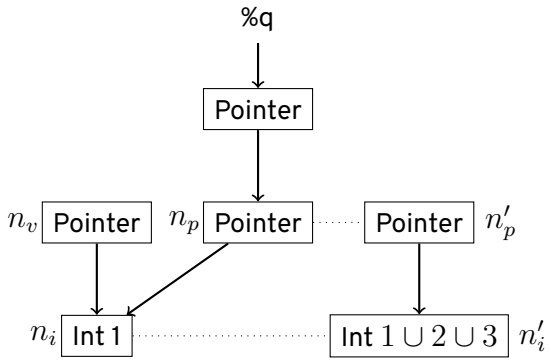
```
1: for all parameters n_p do
2:    N_p := immediatePointees(n_p)
3:    for all n'_p ∈ N_p do
4:       mark n'_p read and escaped, if not this
5:       N_reach := Reachable(n'_p)
6:       mark n_r ∈ N_reach read and escaped, if n'_p not this
7:    end for
8: end for
```

```
1: for all n such that n is reachable from %r do
2:    if isThis[n] then
3:       mark n read
4:    end if
5:    if (n, n_t) ∈ E_This then
6:       mark n_t read
7:    end if
8: end for
```

Figure 5.9: Algorithm for marking nodes read and escaped at top-level returns.



Then, the analysis performs weak updates and adds new weak edges:



In particular, the analysis sees the weak edge involving n_p and n'_p, which causes it to create a new weak edge involving the pointees of n_p and n'_p—that is, from n_i to n'_i. Because those nodes are linked, the analysis also widens the int $2 \cup 3$ node to include the weak update result 1.

We formally express the rules for a weak update following an update out of n_p as follows. For all n'_p such that $(n_p, n'_p) \in E_{\text{Weak}}$, let n_i, n'_i satisfy $(n_p, n_i), (n'_p, n'_i) \in E_{\text{Pointer}}$. Then

$$\begin{aligned} E'_{\text{Weak}} &= E_{\text{Weak}} \cup (n_i, n'_i); \text{ and} \\ n'_i &\text{ is widened to include } n_i. \end{aligned}$$

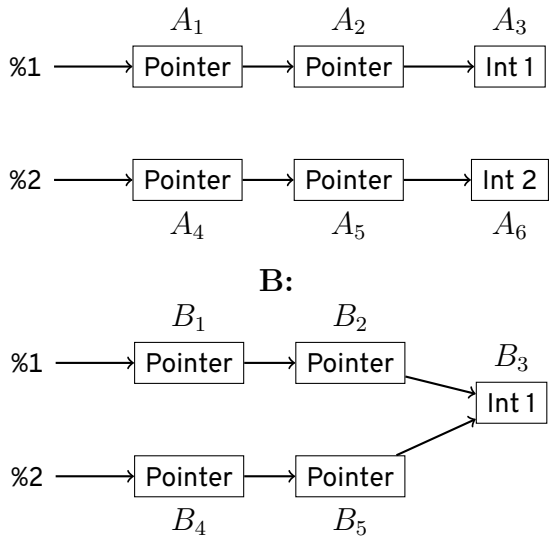
If n_i is a pointer node, then we recursively update n_i and its counterparts.

5.2.7 Merge Operation

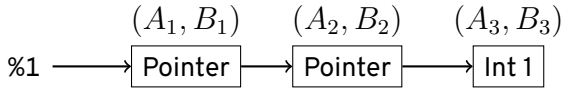
Our merge operation takes two graphs A and B as input and produces graph R as output. A sound merge operation must produce a graph R that represents at least all of the concrete heaps that the input graphs A and B represent. Although pruning could conceptually occur anywhere in the analysis, we implement pruning as part of our implementation of the merge operation—we eliminate nodes that are no longer reachable from LLVM variables.

We first illustrate merging by means of examples. Consider these graphs:

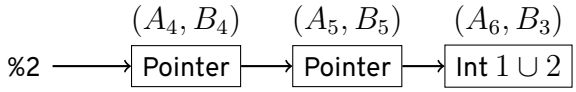
A:



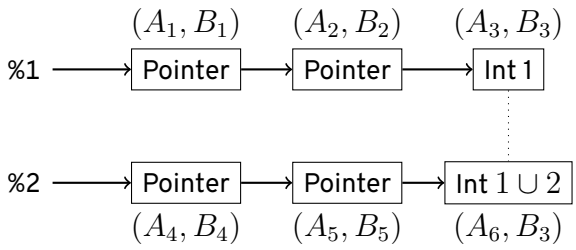
We create merged graph R by simultaneously traversing graphs A and B starting at each LLVM value and creating sets of equivalent nodes. Starting at node %1, our analysis would create the path:



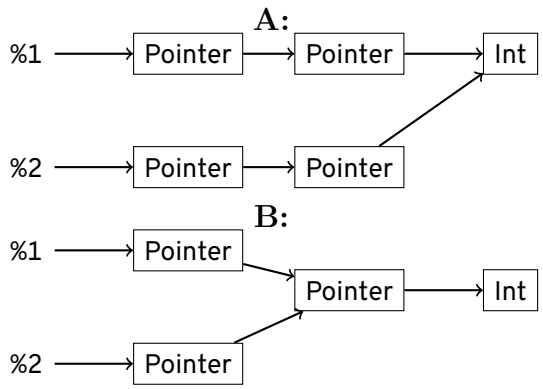
which associates A_n and B_n in the obvious way. (If the paths are not of the same length, our merging algorithm will first equalize them.) All integer nodes contain the union of both ranges. Starting at %2, our analysis would also create the path:



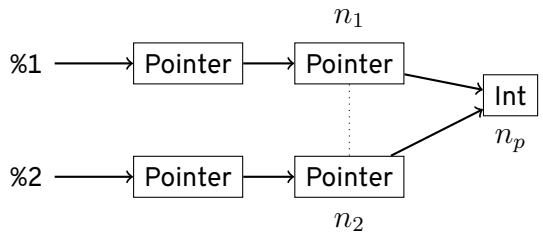
widening the node at (A_6, B_3) to account for both possibilities. Finally, the analysis searches for nodes that belong to more than one set, and links them with weak edges. In our example, B_3 belongs to two sets, prompting the addition of a weak edge. The analysis yields:



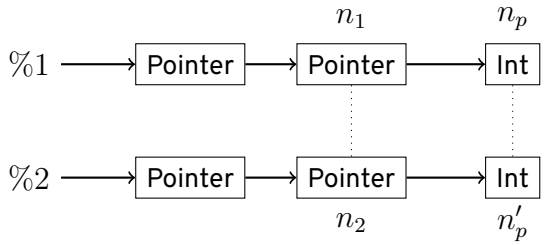
Node splitting. Recall that a weak edge between pointer nodes n_1, n_2 and pointer edges (n_1, n_1') and (n_2, n_2') imply the presence of a weak edge between nodes n_1', n_2'. Our merging algorithm therefore splits nodes if $n_1' = n_2'$. Consider, for instance:



The first phase of the merge yields R as follows:



The second phase of the algorithm therefore searches for pointer nodes n_1, n_2 which point to a common object n_p, such that $(n_1, n_2) \in E_{\text{Weak}}$. It adjusts the graph by making a fresh copy n_p' of n_p and setting n_2's pointee to the copy, linking n_p and n_p' with a weak edge:



Formalization of merge operation.

First, note that E^A and E^B have the same sets of paths starting at local variables, if we add nodes when necessary. We formalize this property as follows.

Let E^A and E^B be sets of edges for graphs A and B and let v be an LLVM variable. Let $(v, n_v^A) \in E^A_{\text{Var}}$ and $(v, n_v^B) \in E^B_{\text{Var}}$. Let path p^A start at n_v^A in E^A. Let p^A take the sequence c of choices of indices at struct nodes. (At pointer nodes, there is only one choice.) Then, we can construct a corresponding path p^B starting at n_v^B in E^B which takes the same choices c. (Recall that a missing edge in our abstraction is equivalent to edges to all nodes of appropriate type in the global alias set; we add edges to nodes in the global alias set as needed.)

Equalization phase. Our algorithm starts by equalizing the lengths of all paths along pointer and structure edges. For each local variable v, it constructs the set of maximal-length acyclic paths in $E^A_{\text{Pointer}} \cup E^A_{\text{Struct}}$ and $E^B_{\text{Pointer}} \cup E^B_{\text{Struct}}$. If corresponding paths have different lengths because path p terminates at node n while p' does not, then the algorithm extends p' by adding edges from its last node to the global alias set.

Labelling phase. The algorithm assigns a unique label to every reachable node in N^A and N^B.

Copying phase. Next, the algorithm creates graph R by copying the corresponding equalized paths from E^A and E^B. It associates with each node $n_R \in R$ the labels from its source nodes (i.e. nodes in A, B reachable with the same choices taken to reach n_R) in N^A and n^B. The contents of n_R is the least upper bound of the contents of its source nodes. The algorithm also copies other edges from the source graphs, including pointer and struct edges which induce cycles, but does not visit them, to ensure termination.

Weak edges phase. The analysis then adds weak edges between nodes in R with the same labels, reflecting the fact that such nodes may alias.

Splitting phase. Finally, the analysis searches for weak edges between pointer nodes which share a pointee. It splits such nodes, erasing one of the pointer edges and replacing it with an edge to the copy. It then creates a weak edge between the split node and its new copy.

5.2.8 Assumptions

As is the case with many static analyses, our analysis is subject to assumptions about the programs under analysis. In our case, we expect that our analysis will identify logically `const` implementations as long as:

1. method arguments never alias any objects reachable from `this`;

2. no part of the heap reachable from `this` may alias any other part of the heap reachable from `this`; and,

3. implicit flows of information do not invalidate logical `const`-ness.

We encode the fairly mild aliasing requirements in how we set up our heap abstraction. Without them, our analysis would report too many false positives. We chose to support only explicit information flows because they suffice for reasoning about the caching implementations that we are most interested in; using implicit flows opens the potential for many more false positives.

For point 2 consider the following:

```
class C {
  mutable bool cached;
  mutable int value;
  bool *internal;
public:
  int getValue() const {
    ... // proper caching
  }
  void unrelated() const {
    *internal = ...;
  }
}
```

Without point 2 we would have to conservatively assume the two boolean fields may alias prior to any call. In `unrelated` the analysis would assume `internal` may point to `cached`. With proper caching the write to `value` within `getValue` would be protected by `cached` being true. If we analyzed `unrelated` with this state, it would set `cached` to unknown while `value` is still read. This would give us a false positive on the next call to `getValue`.

5.3 Results

We implemented our analyses in the LLVM compiler framework and investigated abstract immutability on code from 4 sizeable open-source benchmarks. Our benchmarks are a subset of the benchmarks from Chapter 4. We use the following projects: libsequence, fish shell, Ninja, and Mosh.

We leveraged our framework in Chapter 4 to get information regarding the build process of a project. This information includes all linkage information required to build a library or executable. Given a library or executable, our tool recompiles all the source to LLVM bitcode. It then links all the output together to create an LLVM bitcode version of the library or executable. Our tool analyzes this final linked LLVM bitcode.

We summarize our results at the end of the section. Table 5.1 presents our findings.

5.3.1 libsequence

We analyzed 28 classes from libsequence 1.8.7. Our abstract immutability analysis identified 5 true positives which contain invalid stores, and 4 escaped returns. We found libsequence extensively uses caching. In one case, it misses writing to a cache guard in one branch. This missed write invalidates the caching scheme for the class, resulting in 5 reported writes. Listing 5.3 shows a pruned version of the offending method (we removed one of the writes for brevity).

The `DepaulisVeuilleStatistics` method is responsible for caching the values of the `_DVK` and `_DVH`, and guarding the writes to these fields with `_CalculatedDandV`. If `_NumPoly != 0`, then the caching behaves as expected. When the values are not cached,

Listing 5.3: libsequence's `DepaulisVeuilleStatistics` method misses a cache guard write when `_NumPoly == 0`.

```
1  void DepaulisVeuilleStatistics() const {
2    if (!(rep->_CalculatedDandV)) {
3      if (rep->_NumPoly == 0) {
4        rep->_DVK = 1;
5        rep->_DVH = 0.;
6        return;
7      }
8      else {
9        rep->_DVK = /* ... */;
10       rep->_DVH = /* ... */;
11       rep->_CalculatedDandV = 1;
12     }
13   }
14 }
```

the code begins execution from line 9. Lines 9 and 10 writes the values, then line 11 sets that the values are cached. Any subsequent calls would not write on lines 9 and 10 since `_CalculatedDandV` is true. Otherwise, if `_NumPoly == 0`, every call to this method writes to `_DVK` and `_DVH`. Our analysis found writes to these fields after they were read by the caller of another method (that method returns `_DVK`). In this case, every call recomputes the values, negating any benefit of caching.

We modified the code to correctly write 1 to `_CalculatedDandV` after line 5, and our analysis no longer reported any errors. To libsequence's credit, we later manually inspected the code and found many correct usages of caching. Listing 5.4 shows one such example.

In Listing 5.4, all code that uses `_walls_B`, `_walls_Bprime`, or `_walls_Q` must check `_calculated_wall_stats` before calling `WallStats`. There are no missing checks in the code base. However, one could imagine a new method forgetting this check. This would either remove the performance benefits of caching, as it would have to recompute the values, or be possibly incorrect. It would be incorrect if there was no call to `WallStats` before using, for example, `_walls_B` as it would not have been initialized. We verified that our analysis reports errors when we artificially injected faults by removing the call to `WallStats`, and removing the guard check.

Listing 5.5: libsequence allows mutable access to internal data in 4 methods.

```
double * Sequence::GranthamWeights2::weights() const { return __weights; }
double * Sequence::GranthamWeights3::weights() const { return __weights; }
double * Sequence::Unweighted2::weights() const { return __weights; }
double * Sequence::Unweighted3::weights() const { return __weights; }
```

The 4 escaped returns we found are shown in Listing 5.5. These 4 methods clearly expose internal data. The developers declare the `__weights` field as `mutable double [2]`, and `mutable double [6]` for `GranthamWeights2` and `GranthamWeights3` respectively.

Listing 5.4: libsequence correctly using caching, checked by our abstract immutability analysis.

```
unsigned PolySNP::WallsBprime() const {
  if (rep->_calculated_wall_stats == false) {
    WallStats();
  }
  return rep->_walls_Bprime;
}

void PolySNP::WallStats() const {
  if (/* ... */ ) {
    rep->_walls_B = /* ... */;
    rep->_walls_Bprime = /* ... */;
    rep->_walls_Q = /* ... */;
  }
  else {
    rep->_walls_B = /* ... */;
    rep->_walls_Bprime = /* ... */;
    rep->_walls_Q = /* ... */;
  }
  rep->_calculated_wall_stats = true;
}
```

Both of these fields carry an "logically const" comment. `Unweighted2` and `Unweighted3` are similar.

As well as `weights()`, the `GranthamWeights2` and `GranthamWeights3` classes have a `calculate` method. The calculate method computes a result from its inputs and writes the result to the mutable `__weights` field. It does not cache or protect the values at all. The `weights` method subsequently returns the `__weights` array as a pointer, also not protecting anything. Our immutability analysis detects that the `calculate()` and `weights()` methods work together to violate logical immutability, and reports a `const` violation. The `calculate` method for `Unweighted2` and `Unweighted3` are similar, but they unconditionally write constant values to array elements.

5.3.2 fish shell

We ran our analysis on 56 classes from fish version 2.5.0. Our analysis reported 2 writes and 1 unknown call from fish. The 2 reported writes were interesting. At first glance we believed there was a bug. However, on deeper inspection we found that the caching implementation was correct, but difficult to reason about. The writes are shown on lines 17 and 25 in Listing 5.6.

In Listing 5.6 the initial call to `parser_t`'s `get_lineno` accesses the element at the end of its `execution_contexts` field and calls its `get_current_line_number()` method. This method in turn calls `line_offset_of_character_at_offset`, which then writes to `cached_lineno_count` on lines 17 and 25. The call chain completes and the caller reads the `cached_lineno_count` field.

On subsequent calls our analysis cannot conclude that neither branch can execute. The conditional for neither branch to execute is `cached_lineno_offset == offset`. On the initial call this is satisfied on line 27. However, it is not satisified on line 19. If line 19 was changed to `cached_lineno_offset = offset` instead, our analysis could conclude that property holds on both sides of the branch.

To manually verify this code, the comment indicating why `cached_lineno_offset` isn't `offset` is critical. The write on line 17 is in a loop that iterates over a string, terminating the loop on a null character. If `offset` was beyond the length of the string, that would trigger undefined behaviour. So, instead of being beyond the length of the string, `cached_lineno_offset` will be the index of the null character in the edge case that `offset` is beyond the length of the string. Subsequent calls would not execute line 17, since the character at `cached_lineno_offset`'s index is the null character. For our analysis to verify this method, it would need to understand the layout of C strings, and have a special case for `offset` being beyond the length of the string. This is outside the scope of our analysis.

The sole unknown call for fish is in `history_item_t`'s `matches_search` method. This method calls `find` on a string field to search for an argument substring, and returns whether it was found. Again, we do not explicitly handle C++ standard library functions.

We found fish to be the most conservative of all the benchmarks using `const`. Looking deeper at fish, we found 12 methods that return fields. However, all of the return types for these functions are properly `const` qualified.

Listing 5.6: fish correctly using caching. Our abstract immutability analysis could not successfully check this usage due to complex string properties.

```
int parser_t::get_lineno() const {
  // ...
  lineno = execution_contexts.back()->get_current_line_number();
  // ...
  return lineno;
}
int parse_execution_context_t::get_current_line_number() {
  // ...
  int line_offset = this->line_offset_of_character_at_offset(/* ... */);
  // ...
}
int
parse_execution_context_t::line_offset_of_character_at_offset(/* ... */) {
  // ...
  if (offset > cached_lineno_offset) {
    // ...
      cached_lineno_count++;
    // ...
    cached_lineno_offset = i;
    // note: i, not offset, in case offset is
    //       beyond the length of the string
  }
  else if (offset < cached_lineno_offset) {
    // ...
      cached_lineno_count--;
    // ...
    cached_lineno_offset = offset;
  }
  return cached_lineno_count;
}
```

5.3.3 Mosh

We ran our analysis on 48 classes from Mosh version 1.2.6. Of these classes, we found 1 with an escaped return and 3 where the state of `this` reaches an unknown call.

Listing 5.7 shows the invalid escaped return. `AlignedBuffer`'s `data` method (which is `const`-qualified) returns a plain `char *`, allowing the caller to modify internal data through the returned pointer.

Listing 5.7: Mosh returning non-`const` qualified pointers to `AlignedBuffer`'s internal data.

```
char * AlignedBuffer::data() const { return m_data; }
```

The 3 unknown calls were all false positives. The first two stem from the `compare` method in `Complete`. At some point this method writes fields to a local string buffer and compares the string's content. LLVM does not provide our analysis with the implementation of string's `append` method, so our analysis conservatively assumes the field arguments may be written to. The final unknown call stems from the `diff_from` method in the same `Complete` class. This method creates a temporary object from values of fields and explicitly calls operator `delete`, which our analysis does not understand.

Our immutability analysis produced false positives in 3 cases. All of them are essentially due to not having enough information about standard library functions and conservatively overapproximating their behaviour.

5.3.4 Ninja

We ran our analysis on 20 classes from Ninja version 1.7.2. None of the classes from Ninja generated any results due to store, or unknown calls. However, our analysis found 4 instances of fields that escaped through the return value without a fully `const` qualified return type. Listing 5.8 shows these methods, along with their bodies.

Listing 5.8: Ninja has 4 classes that return non-`const` qualified pointers to internal data.

```
const vector<Node*>& DepsLog::nodes() const {
  return nodes_;
}
DepsLog* ImplicitDepLoader::deps_log() const {
  return deps_log_;
}
DepsLog* DependencyScan::deps_log() const {
  return dep_loader_.deps_log();
}
BuildLog* DependencyScan::build_log() const {
  return build_log_;
}
```

First, `DepsLog::nodes` should instead return a vector of `const Node*` to be transitively immutable. The remaining 3 methods should `const` qualify the object type to be transitively immutable. Otherwise, for example, the caller of `deps_log` may mutate the state of

a `const` qualified `ImplicitDepLoader` through the returned value. All of these methods expose internal state.

5.3.5 Summary

Table 5.1 summarizes our results. Our analysis checked 152 classes across 4 projects. Out of the 18 warnings generated by our tool, there were 6 false positives. 2 of the false positives in fish were related to writes. Manual inspsection of these writes found they were indeed valid, but required complicated reasoning about strings to check. The remaining false positives stem from 5 calls to standard string methods, and one explicit call to the `delete` operator.

Table 5.1: Open-source benchmarks largely use abstract immutability correctly; immutability errors in practice are usually representation exposure.

	Classes analyzed	Escaped returns	Unknown calls	Writes	False positives	Runtime
libsequence 1.8.7	28	4	0	5	0	2h 57m 53s
fish 2.5.0	56	0	1	2	3	6m 37s
Ninja 1.7.2	20	4	0	0	0	11s
Mosh 1.2.6	48	1	3	0	3	3h 42m 44s

Our analysis generated a low number of false positives. We believe our assumptions were needed to make our tool useable. The runtime is within an acceptable range: we do not expect the analysis to run for every project compliation. For fish, the write false positive, while it was incorrect, did indicate a complicated method which could be very brittle.

Chapter 6

Related Work

We discuss 3 major categories of related work: types, language features, and analyses. For types we explore both type systems and type inference. Some of these type systems, along with inference, support further analyses. There is great potential for future work to study the benefit of language features in general. In this work we aim to show the benefit of immutability. Our work uses both dynamic and static analyses to enforce abstract immutability. We leverage existing techniques for both dynamic and static analysis, and compare our work to related analyses.

Types include type system extensions to Java similar to `const`. The Java programming language has no exact analog to C++'s `const` operator. Related work defined immutability annotations for Java and statically and dynamically verified that programs satisfy their annotations. Potanin et al. [61] provide a recent discussion of immutability terminology, consistent with our description in Chapter 2, and compare research implementations in depth. Type systems for immutability often include type inference, which automatically finds types that should include, for instance, a `const`-like keyword. Reference immutability in Java has not yet been added to the Java language specification; authors of related work augment un-annotated programs.

We believed it was important to show that developers benefit from immutability declarations. Prior to this work, there was no hard evidence that developers used immutability in practice despite a pervasive belief on the part of language implementations and much work in this space (as discussed in this chapter). Chapter 4 is partly inspired by studies of other language features. We also explore whether developers use abstract immutability in Chapter 3. Other related work improves the flexibility of immutability for developers. Note that both inference systems and static analysis can improve usability by helping developers add immutability declarations.

Our dynamic analysis implementation in Chapter 3, at its core, verifies that C++ programs satisfy a strengthened version (deep concrete immutability) of their `const` annotations. In Chapter 4 we use static analysis to find methods that obey deep concrete immutability. Our analysis in Chapter 5 verifies abstract immutability: it permits writes that do not modify a caller's view of an object. Other proposed analyses verify whether (Java) methods are pure, i.e. have no visible side-effects; there is a strong connection between purity and immutability [74]. Some purity analyses build on top of proposed type systems.

6.1 Type Systems

Researchers have developed type systems that extend Java to add reference immutability. Reference immutability ensures that objects accessed through a reference cannot be modified. Typically these systems ensure deep concrete immutability. Object immutability, on the other hand, ensures that the object itself does not change. Reference immutabilityis similar to `const` references or pointers in C++, while object immutability is equivalent to having a class with only `const` methods (and no public fields). We chose to study C++ because it's widely used in practice along with its existing `const` annotation, which enables us to analyze code in the wild. Reference immutability in Java is not standard; authors of related work augment un-annotated programs.

6.1.1 Javari

Javari [83, 82] adds reference immutability to Java. The authors extend Java's type system and add additional run-time checks using the Checker framework [59]. Their type system introduces 4 new keywords to Java: `assignable`, `readonly`, `mutable`, and `romaybe`.

Java's `final` does not allow a reference to change after its declaration. The `assignable` keyword complements the `final` keyword by allowing the reference to change in any context. The other keywords control the mutability of the object pointed to by the reference. The `readonly` keyword indicates that the reference can only use `readonly` object methods.

Javari's cited improvements over C++ are as follows:

- Does not allow unchecked casts;
- Prevents misuse of the type system (unions and varargs);
- Supports multi-dimensional arrays;
- Allows parameterization of code based on variable immutability;
- Ensures pointers are transitively `const`.

Since Javari does not have an underlying C++ `const` specification to build on top of, it has to implement all of those checks itself. In terms of what we check, our work in Chapter 5 matches Javari in terms of transitivity and downcasts. Our work in Chapter 3 also reports writes to mutable fields at runtime.

Unlike Javari, we investigate the behaviour of a set of real-world benchmark programs, developed against the C++ `const` semantics (and which, by necessity, satisfy those semantics). Our empirical study therefore points out the difference in practice between `const` semantics as they exist—shallow immutability, mutable, and `const` casting; and a stronger version of these semantics—deep immutability, no mutable, and no const casting.

Javari aims to ensure that a `readonly` (effectively, deep concrete immutable) typed object does not mutate its state or any state transitively reachable through its references. Javari, like C++, includes a `mutable` keyword, which allows developers to specify that a field may be modified regardless of `readonly` qualifiers. Javari also inserts dynamic checks to verify that downcasts maintain the immutability qualifier of the type. Essentially, Javari

provides a safer version of C++'s `const` that, due to the nature of Java, maintains deep immutability unless the developer explicitly opts out of the checks.

Discussion. Javari keywords behave similarly to C++'s `const` and `mutable`. When `const` applies to a primitive type in C++, it behaves the same as `final`. A lack of `const` qualifier behaves the same as `assignable`. The keyword `romaybe` allows Javari to template methods over mutability. Any method using `romaybe` compiles to two versions: a `mutable` version and a `readonly` one. A disadvantage to this approach is that `romaybe` is an additional keyword for developers and is often needed, adding noise to the source code. For instance, a `get` method needs `romaybe` on the `this` parameter allow its use on a mutable reference.

Javari's type system has many improvements over C++. In Chapter 3, we propagate `const` similar to Javari's default behaviour of `readonly` (deep concrete immutability). The writes-through-`const` we report would require the `mutable` keyword in Javari. We found that, in our benchmarks, most writes were explicitly allowed by developers, which Javari cannot check.

Our analysis in Chapter 4 checks a subset of methods that could be `readonly` in Javari. In Javari, if a method is `readonly` it must have deep concrete immutability. C++, provides no similar guarantees.

The key difference between our approach in Chapter 5 and Javari is that we use a static analysis to ensure that writes do not affect the visible state of an object, compared to Javari's approach of marking fields as `mutable` and allowing arbitrary access. Checking abstract immutability properties requires in-depth analysis to ensure the visible state of the object is not modified.

6.1.2 ReIm

ReIm [36, 17], by Huang, Milanova, Dietl, and Ernst, is another framework which adds reference immutability to Java. ReIm is intended for purity inference, and can therefore use a simpler type system, omitting parts that are not needed for purity inference. For instance, references may never be modified through a `readonly` reference, and there is no `assignable` or `mutable` that allows fields to change in a `readonly` method. The type system also differs in that ReIm's system is context-sensitive. Instead of `romaybe` in Javari, ReIm introduces `polyread`. The immutability of references returned using `polyread` match the qualifier of the caller. This allows methods to vary over immutability without having to create two versions of the same method. The intention of ReIm is to ensure methods are side-effect free (pure).

ReIm uses a viewpoint adaptation from Universe Types [18]. They combine the declared type of a method parameter and the type of the parameter used in a call to determine the type from the caller's point of view. This ensures that any returned types have the correct immutability type qualifiers (always preferring `readonly` if possible).

Discussion ReIm's type system, unlike Javari, cannot handle caching as there is no `mutable` or `assignable` keyword. The simpler type system suffices for purity—caches are

impure. Javari also supports annotations on type arguments for parametric classes, while ReIm does not.

6.1.3 Glacier

The Glacier system by Coblenz et al [13, 14] implements transitive class-based immutability in Java, and justifies its immutability design with a significant case study, but no empirical numbers (unlike our results in Chapter 4). In Glacier, an object that instantiates a class declared as `@Immutable` will have all fields immutable. Furthermore, any transitively reachable fields must also be immutable. Glacier's immutable objects are completely immutable; Glacier does not contain the notion of a "mutable field" and prohibits transitive writes for immutable classes.

Any class developers add `@Immutable` to is checked by the type system. The authors added their immutable annonations, and rules, using the Checker framework. Any fields or methods that are not immutable are shown to the developer as a compile-time error.

Discussion. The Glacier system assists developers writing immutable objects. It does not have a notion of reference immutability: all classes must be written with deep concrete immutability. We believe this work is useful for Java developers. In Java, writing an immutable object requires making every field is private and `final`. This requirement applies to any objects referred to by fields as well, but developers may miss those fields. This work, through its user study, shows a vast improvement for developers writing immutable classes over using `final`. Glacier does not allow for reference immutability or abstract immutability.

6.2 Type Inference

Both Javari and ReIm have corresponding inference systems. However, neither of these systems can check abstract immutability, our focus in Chapter 5. They take existing source code and analyze it to add type qualifiers such that it type checks.

In C++, the `const` qualifier serves two roles: it is a type qualifier (for fields and local variables) as well as an annotation for methods. Foster et al. [23] inferred `const` type qualifiers for C programs and found that their case studies could have included many more `const` annotations than they did. More recently, Greenfieldboyce and Foster [27] presented a technique for inferring type qualifiers for Java using their JQual tool. They apply JQual to inferring the `readonly` qualifier, a variant of the version of `readonly` proposed by Javari [83]. Both of these related works use a type systems approach to propagate type constraints through the program. We focus on the method-annotation role of `const`, which is closer in spirit to determining whether a method is pure or not. Our approach in Chapter 4 uses a simple dataflow analysis rather than a type systems approach, and we work at a per-method granularity, determining whether each method should be `const` or not (optionally using the assumption that `const` methods do not mutate).

6.2.1 Javarifier

Javarifier uses the type system described in Javari [62], but renaming `romaybe` to `polyread`. Their analysis is built on Soot [45]. Their analysis is flow and context insensitive for inferring `readonly`. For inferring `assignable` and `mutable` they use heuristics. Their 2 heuristics are as follows:

1. Methods annotated as `readonly` are correct, so any modifications do not change the abstract state.

2. Fields that are only used in a single method or fields that are declared as `transient` are not part of the state (they don't serialize).

Discussion. These heuristics are simple and do not capture many uses that would be considered abstractly immutable. For instance, they can not handle caching or logging. Our analysis in Chapter 4 is similar. However, our analysis in Chapter 5 could be extended to maximize the set of immutable methods. This would allow inferring abstractly immutable methods.

6.2.2 ReImInfer

ReImInfer [36, 39, 38, 37] uses the type system described in ReIm, which was designed to find method purity. Their type inference analysis is precise, scalable, and requires no manual annotations. For each reference type in the program they find the most restrictive type qualifier possible. They found their runtime almost scaled linearly while being as precise as Javarifier.

ReImInfer defines a "best typing" policy. They found a lexicographic ordering of types from most restrictive to least restrictive produced the best results. They implemented their inference system using the Checker framework and found that their system is precise and scalable.

ReIm and its corresponding type inference system, ReImInfer [36], are similar to Javari, except with a context sensitive type system. Its type system allows the immutability of the return type to match that of the calling reference. This allows methods to be reused without requiring mutable and read-only versions. ReImInfer is a type inference system that maximizes the amount of methods marked as `readonly`. The authors of ReImInfer report that 41–69% of methods can be marked as `readonly`.

Discussion. Aside from the type system, their analysis is similar to our analysis in Chapter 4. The major difference is that their type qualifier depends on the current context. This is an improvement over C++ where developers need to write two versions of the method. However, we found the number of immutable methods was higher than ReImInfer's since we include more than deep concrete immutability.

Some developers do not intend deep immutability for data structures. For instance, some methods return an immutable list of mutable objects. Our results agree with ReImInfer's, we found that developers do miss many immutable methods. Overall, we found developers annotate the majority of the immutable methods without any assistance.

6.2.3 Inferring `const` for C programs

There are a number of generic frameworks for inferring user-defined type qualifiers for programming languages [23, 9]. We investigated Foster, Fähndrich, and Aiken's [23] framework since they demonstrate their framework by creating a `const` inference system for C programs. Their system infers the maximum number of `const`s that can be present in a program. They found, for programs that already used `const`, there could be up to 2.5 times more `const`s than currently present in the code.

C/C++ uses a monomorphic type qualifier system, while their proposed type qualifier system uses polymorphic type qualifiers. A monomorphic type inferencer system adds `const` if it's possible, regardless of the context. Note that monomorphic type qualifiers causes the problem of repeated functions. For instance, in C++, there needs to be two versions of `strchr`. There's no way to express that the `const` qualifier of the input type should match the returned type. They found 5-16% more `const` annotations could be added to C functions using their polymorphic analysis.

Discussion. Since their inference system operates on C programs, they must make conservative assumptions due to loopholes in C (as discussed in Chapter 2.2.5). Their polymorphic system also would correctly type `strchr`, since the type of a possible return value (`return NULL;`) would be seen as unrelated in their system. In Chapter 4 we ensure that if a field is returned, the return type is also immutable. This allows us to infer that two versions of the same method are both correct.

The definition of `const` the authors use is shallow concrete immutability. Since their analysis is for C programs, there is no concept of `mutable`, `const_cast` or member functions, and thus no abstract immutability. They also do not verify polymorphic relationships. For instance, two type qualifiers could both vary with respect to `const` within a function but be unrelated.

6.3 Language Features

We believe it is important to show immutability is useful for developers. There are no previous studies specifically on immutability, however there are existing studies for other language features. For immutability, there are proposals for improving the usability of immutability: frozen objects and explict immutable annotations for immutable objects.

6.3.1 Usability of Language Features

Our work empirically surveys existing codebases to explore developers' use of language features—in our case, C++ and the `const` qualifier. Related work in this area is rare. Richards et al. [64] surveyed deployed JavaScript code to understand how the `eval` keyword is used in practice, and characterized the uses that they found. In the Python world, Holkner and Harland investigated the use of dynamic features in that language [35].

Discussion. Our work shares with theirs the desire to understand code as it exists in the wild. A key difference between our work and theirs is that `const` could be elided without any immediate implications on software behaviour. (We presume that this elision would make long-term maintenance more difficult). On the other hand, `eval` makes code maintenance more difficult.

6.3.2 Frozen Objects

Leino, Müller, and Wallenburg [49] have proposed frozen objects, a flexible version of immutable objects. Frozen objects are supported as part of the base language in Ruby. This system allows a mutable object to become an immutable object by applying the `freeze` operator. After an object becomes frozen, it is a transitively immutable object. This system is dynamic, all checks are done at run-time. However, the authors mention this system applies to static type systems which support ownership transfer.

Frozen objects allow for multi-stage initialization. Immutable objects usually require all writes to occur in the constructor, and such a scheme is not possible if two objects have a direct dependence on each other.

Discussion. While frozen objects are more flexible than immutable objects, we believe reference immutability has similar flexibility (although for different purposes). However, under reference immutability developers would have to ensure that, after an object's initialization, the only references to the object are immutable. It is not clear whether or not frozen objects are more usable for developers. Leino, Müller, and Wallenburg [49] do not include any studies of their system's usability.

6.3.3 Abstract Machine

Chisnall et al. [10] propose a memory-safe abstract machine for C, and identify programs relying on implementation defined behaviour for correctness. The main focus of the work is pointer arithmetic. However, one property from the C standard is that immutable objects (declared with `const`) are never mutated through non-`const` aliases. They identified static source locations which removed a `const` qualifier. Later they enforced `const` at runtime. In the end the authors disabled `const` enforcement by making it advisory.

Discussion. The abstract machine run-time resembles our approach in Chapter 3 of using shadow values to track the declared `const`-ness of an object. (They do not investigate transitive immutability.) However, their runtime would halt on writes-through-`const`, while we report them. The authors noted they had to disable this run-time enforcement. All of their subject programs removed a `const` qualifier at some point. It was beyond the scope of their work to investigate how and why their subject programs removed the `const` qualifier. They also statically checked source locations that removed `const` (through modifying Clang). The number of static locations was high, and the authors did not comment on them.

6.4 Dynamic Analysis

Our dynamic analysis approach is similar to approaches used in Umbra [94] and Dr. Memory [8]. Like Umbra and Dr. Memory, we use shadow memory to detect interesting program behaviours. However, our ConstSanitizer approach builds directly on LLVM and does not use a dynamic instrumentation platform. We follow the sanitizer family of tools in Clang including AddressSanitizer [71] and MemorySanitizer [73].

Discussion. Like our tool, the sanitizer tools generate additional code and interact with lightweight run-time libraries. Furthermore, the properties that we verify in Chapter 3 are novel `const`-related properties, compared to Dr. Memory, which looks for memory errors such as accesses to unallocated space, and Umbra, which helps developers understand threads' memory access patterns and implements almost-free custom watchpoints. Unlike the other tools, violations of our property is not necessarily an error. We use our reports to better understand how developers use immutability.

6.5 Static Analysis

To our knowledge, our work in Chapter 5 is the first to reason about abstract immutability in the presence of writes to all parts of an object. There has been much previous work on immutability, but that work typically excluded some parts of the objects (i.e. certain fields) from the analysis. Our class-based approach, by contrast, allows writes to object fields, as long as the writes do not change a previously-exposed part of an object. We compare our work to previous work in more detail below.

Our work also uses some notions of information flow; however, it avoids the limitations inherent in a purely information-flow approach. Of course, our work relies on results from pointer analysis, and we situate our use of pointer analysis within the context of that area of research.

Other analyses look for pure functions. Pure functions are functions with no side effects. For object immutability, this definition is very restrictive, side effects may be desired. Developers could still benefit from the knowledge that the underlying object itself did not change. The tools we discuss analyze methods that use a less restrictive version of purity, allowing new state. Note that our abstract immutability definition applies to objects only.

A pure method does not modify any state accessible before the method was called. Pure methods may create and modify objects to return to the caller. A function which writes to no global state and has all arguments transitively `const` is pure.

Our analysis in Chapter 4 identifies a limited subset of those that would be identified as pure by typical purity analyses, e.g. [68, 36]. Many pure methods do more than is allowed by our analysis; the most significant difference is that a pure method may write to parts of the heap that are freshly-allocated by that method. Our analysis in Chapter 4 is also quite conservative in declaring methods to be `const`-able and will reject any writes; more sophisticated analyses will allow writes to unrelated parts of the heap.

6.5.1 Stationary Fields

Unkel and Lam developed a tool which analyzed Java code for stationary fields [84]. A stationary field is a field that only has writes to it before reads. A Java `final` field can only be written to in a constructor. By contrast, a stationary field is only written to before it is read. In essence, a stationary field acts like a `final` field but with fewer restrictions on where initialization may occur. They found that stationary fields are more common than `final` fields (44–59% vs. 11–17% respectively). This indicates that, for their benchmark programs, a majority of the fields are stationary.

Their static analysis uses a summary-based algorithm. Their analysis is flow-sensitive, context-sensitive and interprocedural. They track when an object is "lost", that is when some other object may point to it. For a field to be stationary, all reads must happen after writes before the object is lost, and lost objects may only have field reads.

Nelson et al. [57] performed a follow-up study using a dynamic analysis. That study found that 72–82% of fields in Java programs are stationary. Their work, like ours, empirically explores how programs use (or could use, in their case) language features.

Discussion. Stationary fields are similar to abstract immutability. However, stationary fields apply to the lifetime of the object: we focus on an object's immutable methods. This allows us to capture objects that vary between stationary and non-stationary several times over the object's lifetime. Since the type qualifiers of active references to objects can change between program points, our system supports programs that cycle between write and read phases. Note that if all references to an object are `const`, the definition of abstract immutability and stationary fields are the same. Unkel and Lam do not mention how many of these stationary fields represent immutable objects.

6.5.2 Escape Analysis and Information Flow

Our analysis builds on the concept of escape analysis [11]. Essentially, escape analysis flags locations that escape from a class that are written to after they escape. Choi's work on escape analysis attempted to detect non-escaping objects, while our work flags escaped values.

Our work shares some goals with information flow analysis [91, 55]. Both our work and information flow analysis aim to prevent the release of certain information. However, a key difference is that the release of information is not sufficient to make an object not abstract immutable; we flag code that releases a value and also subsequently changes the backing location.

6.5.3 JPPA

Java Pointer and Purity Analysis (JPPA) [68, 67] defines a method as pure if it does not mutate any location that exists in the program state right before the invocation of the method. Their analysis also provides useful information for parameters in impure methods. They identify two types of parameters: *read-only* and *safe*. A read-only parameter does not mutate any objects transitively reachable from the parameter within the method. A

safe parameter is a read-only parameter that does not create any new externally visible heap paths to objects transitively reachable from the parameter.

Their analysis is built on top of a combined pointer and escape analysis [86]. Their purity analysis is interprocedural and depends on a complex analysis. Pure methods must not have a reference escape globally and must not mutate any reference's fields. They found that on average half the methods in their benchmark suite are pure. Rytz et al. [66] instead use a simpler flow-insensitive analysis and find similar results.

Discussion. Their analysis requires manual inspection to establish that implementations, such as caching, are pure. Their system also does not rely on type qualifiers and requires the whole program. Our analysis instead focuses on the code at a per-class level, adding correct type qualifiers for our definition of `const`. Later, our analysis could leverage our abstractly immutable methods for more complex whole program analyses (including optimizations).

6.5.4 JPure

JPure [60] defines a method as pure if it does not assign (directly or indirectly) to any field or array cell that existed before it was called. Their tool does purity checking for methods annotated with `@Pure`, `@Local`, and `@Fresh`. For a method to be `@Pure`, it must fit their original definition. Methods annotated with `@Fresh` only return newly allocated objects. To deal with objects such as iterators that only update their own fields, methods may be annotated with `@Local` to ensure only local changes. This allows their system to handle iterators, as `@Fresh` objects that only update their `@Local` state are allowed in `@Pure` methods.

Their static analysis is flow-sensitive, context-sensitive and intraprocedural. They check that every annotation is valid or flag the error to the developer. If code passes their static analysis, the annotations are correct. Note that, because of dynamic dispatch in Java, subclass implementations must have the expected annotations as well.

They also include a purity inference tool, which uses a greedy approach. It assumes all methods are annotated as `@Fresh` or `@Pure` and that all fields are `@Local`. Based of the content of the method or its context in the static class hierarchy [16], the tool removes annotations as needed.

Discussion. Their checker tool attempts to verify their annotations, like ours, which checks type qualifiers. However, since they are concerned about purity, they analyze static methods in addition to object methods at this stage. Our analysis in Chapters 4 and 5 only needs to handle object method definitions and their fields.

6.5.5 Combined Static and Dynamic Analysis

Artzi et al. [4] propose a staged approach for finding reference immutability for Java programs. Their stages are as follows: 1) an intraprocedural static analysis, 2) an interprocedural static analysis, and 3) a dynamic analysis. Their analysis determines the mutability of method parameters. The goal of that work was to scale better and produce more precise results than static analysis alone.

Discussion. Artzi et al.'s work shows an interesting application of dynamic analysis applied to reference immutability. We may choose to add dynamic analysis to our approach to improve precision. Artzi et al.'s definition of purity is similar to that used by JPPA in Chapter 6.5.3.

6.5.6 Pointer Analysis

Because we aim to check abstract immutability of objects, and because our definition of abstract immutability includes the transitive state of objects (through pointers), our analysis must track relatively detailed pointer information. In particular, we use standard pointer analysis techniques [33] to track intraprocedural may- and must-information. The concept of integrating pointer analysis with client analyses is fairly common, e.g. Fink et al.'s typestate verification [22].

To situate our abstraction: it is a field-sensitive, flow-sensitive intraprocedural pointer analysis which tracks may-alias information using weak edges and must-alias information using E_{Seq} edges. We use the global alias set to represent unknown objects (rather than using summary objects). We effectively inline method calls that do not cross a class boundary; exhaustive inlining works because our class-based scope is narrow.

Chapter 7

Conclusion

This dissertation investigated the usage and enforcement of abstract immutability in code. In Chapter 3, we explored how often and why developers write through immutability declarations. We found that, in the cases which were not mistakes, developers principally used abstract immutability. In Chapter 4, we showed that developers strive to add immutability declarations to methods. We found that over 50% of methods in our benchmark suite are immutable, while immutable and all-mutating classes are not common. In Chapter 5, we defined abstract immutability formally. We created a novel static analysis that checks for abstract immutability and does not flag many false positives. We expand on our conclusions below.

Dynamic Observations of Writes-Through-Immutability (Chapter 3). We show, through our dynamic analysis, that the `const` qualifier (indicating immutability) in C++ is extensively used in real-world code, but the developer intent behind `const` usage is unclear. This dissertation presented our ConstSanitizer system. ConstSanitizer dynamically detects writes through `const` qualifiers which are legal in C++, but which modify state transitively starting from a `const` qualifier, write to mutable fields, or write to values whose `const`-ness has been cast away. Our results show that, although writes through `const` are ubiquitous across our 7 C++ and 1 C benchmark programs, there are only a small number (17) of archetypes for these writes. We used our results to develop a classification of writes according to root cause (transitivity, mutable field, or const cast) and attributes (synchronized, not visible, buffer/cache, delayed initialization, incorrect). Our work helps understand how the `const` qualifier is used in practice and leads us to conclude:

- Developers definitely violate bitwise `const`.
- The majority of write-through-`const` archetypes (9/17) are incorrect code which observably change an object's state.
- On our benchmarks, programs write-through-`const` about equally often using transitive writes through fields (8/17) and writes to mutable fields (8/17).
- We observed four classes (N, B, D, S, discussed in Chapter 3.3) of valid reasons for writes-through-`const`-qualifiers. For instance, sometimes developers write-through-

`const` to delay initialization or implement buffer caches. Such writes should be checked by tools.

- About half (9/17) of the observed usages are invalid, consisting of methods which implemented exceptions to an object's const-ness; perhaps the developers chose to add one exception rather than remove const completely.

Static Observations of Immutability (Chapter 4). We created a framework to collect all immutability declarations in a project, along with a conservative static analysis that checks concretely immutable methods. We found that immutable classes are often used among all 7 open software projects we investigated. Across all the projects we found that a median of 7.15% of classes are immutable classes. To the best of our knowledge, these were all written correctly for the definition of immutability proposed in this dissertation. Developers also write all-mutating classes, which should contain only mutating methods. A median 15.5% of a program's classes are written as all-mutating. However, we found that approximately half of these classes omitted `const` annotations on some of their methods. We manually verified they our analysis finds most of the omitted annotations. Using our analysis we found that a median of 22.6% of classes that are eligible to have `const` annotations added should have at least one of their methods `const`-annotated. We intend to release our Immutability Check tool as open-source software and believe that it is already useful for developers.

Abstract Immutability Analysis (Chapter 5). This dissertation presented a sophisticated analysis that is the first to verify abstractly immutable methods. We formally define abstract immutability, and implemented the analysis using LLVM bitcode. Succinctly, a set of methods are abstractly immutable if no locations are written-to after they have become visible to callers. Our analysis uses a novel subsequent call analysis to check abstract immutability properties. We explored the use of immutability in 4 real-world benchmarks and found that our analyses enabled us to find incorrect implementations of abstract immutability in these benchmarks. Without our analysis, it would be very difficult to in vast codebases implementing abstract immutability in sometimes-intricate ways.

Summary. This dissertation explores abstract immutability. We developed tools to: 1) investigate the purposes developers use abstract immutability for; 2) confirm the ubiquity of abstract immutability across codebases; and 3) check abstract immutability via a novel static analysis.

References

[1] Jason Ansel, Cy P. Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman P. Amarasinghe. "PetaBricks: a language and compiler for algorithmic choice". In: *PLDI*. June 2009, pp. 38–49.

[2] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. "JDiff: A differencing technique and tool for object-oriented programs". In: *ASE* 14.1 (2007), pp. 3–36.

[3] Taweesup Apiwattanapong, Raúl A. Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. "MATRIX: Maintenance-Oriented Testing Requirements Identifier and Examiner". In: *TAIC PART*. Aug. 2006, pp. 137–146.

[4] Shay Artzi, Adam Kiezun, David Glasser, and Michael D. Ernst. "Combined Static and Dynamic Mutability Analysis". In: *ASE*. Nov. 2007, pp. 104–113.

[5] Vipin Balachandran. "Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation". In: *ICSE*. May 2013, pp. 931–940.

[6] Mike Barnett, David A Naumann, Wolfram Schulte, and Qi Sun. "99.44% pure: Useful abstractions in specifications". In: *FTfJP*. 2004.

[7] Samual Bates and Susan Horwitz. "Incremental Program Testing Using Program Dependence Graphs". In: *POPL*. Jan. 1993, pp. 384–396.

[8] Derek Bruening and Qin Zhao. "Practical memory checking with Dr. Memory". In: *CC*. 2011, pp. 213–223.

[9] Brian Chin, Shane Markstrum, Todd D. Millstein, and Jens Palsberg. "Inference of User-Defined Type Qualifiers and Qualifier Rules". In: *ESOP*. Mar. 2006, pp. 264–278.

[10] David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. "Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine". In: *ASPLOS*. 2015.

[11] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. "Escape Analysis for Java". In: *OOPSLA*. 1999, pp. 1–19.

[12] David G. Clarke and Sophia Drossopoulou. "Ownership, Encapsulation and the Disjointness of Type and Effect". In: *OOPSLA*. Nov. 2002, pp. 292–310.

[13] Michael J. Coblenz, Whitney Nelson, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. “Glacier: transitive class immutability for Java”. In: *ICSE*. 2017, pp. 496–506.

[14] Michael J. Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad A. Myers, Sam Weber, and Forrest Shull. “Exploring language support for immutability”. In: *ICSE*. 2016, pp. 736–747.

[15] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. “Input-sensitive profiling”. In: *PLDI*. June 2012, pp. 89–98.

[16] Jeffrey Dean, David Grove, and Craig Chambers. “Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis”. In: *ECOOP*. Aug. 1995, pp. 77–101.

[17] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muslu, and Todd W. Schiller. “Building and Using Pluggable Type-Checkers”. In: *ICSE*. May 2011, pp. 681–690.

[18] Werner Dietl, Sophia Drossopoulou, and Peter Müller. “Generic Universe Types”. In: *ECOOP*. Aug. 2007, pp. 28–53.

[19] Will Dietz, Peng Li, John Regehr, and Vikram S. Adve. “Understanding integer overflow in C/C++”. In: *ICSE*. June 2012, pp. 760–770.

[20] Jon Eyolfson and Patrick Lam. “C++ const and Immutability: An Empirical Study of Writes-Through-const”. In: *European Conference on Object-Oriented Programming (ECOOP)*. 2016, 8:1–8:25.

[21] Felix Fang. *Personal communication*. 2015.

[22] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanual Geay. “Effective Typestate Verification in the Presence of Aliasing”. In: *ISSTA*. 2006.

[23] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. “A Theory of Type Qualifiers”. In: *PLDI*. May 1999, pp. 192–203.

[24] The C++ Foundation. *The C++ Super FAQ: Const Correctness*. Apr. 2017. URL: `https://isocpp.org/wiki/faq/const-correctness`.

[25] M. Frigo and S.G. Johnson. “The design and implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231.

[26] Robert M. Fuhrer, Frank Tip, Adam Kiezun, Julian Dolby, and Markus Keller. “Efficiently Refactoring Java Applications to Use Generic Libraries”. In: *ECOOP*. July 2005, pp. 71–96.

[27] David Greenfieldboyce and Jeffrey S. Foster. “Type qualifier inference for Java”. In: *OOPSLA*. 2007, pp. 321–336.

[28] Dan Hao, Tian Lan, Hongyu Zhang, Chao Guo, and Lu Zhang. “Is This a Bug or an Obsolete Test?” In: *ECOOP*. July 2013, pp. 602–628.

[29] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. “On-Demand Test Suite Reduction”. In: *ICSE*. June 2012, pp. 738–748.

[30] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. "Data representation synthesis". In: *PLDI*. June 2011, pp. 38–49.

[31] Christopher M. Hayden, Edward K. Smith, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. "Evaluating Dynamic Software Update Safety Using Systematic Testing". In: *IEEE Trans. Software Eng.* 38.6 (2012), pp. 1340–1354.

[32] Nevin Heintze and Olivier Tardieu. "Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second". In: *PLDI*. June 2001, pp. 254–263.

[33] Michael Hind. "Pointer analysis: haven't we solved this problem yet?" In: *PASTE*. June 2001, pp. 54–61.

[34] Michael Hind and Anthony Pioli. "Which Pointer Analysis Should I Use?" In: *ISSTA*. Aug. 2000, pp. 113–123.

[35] Alex Holkner and James Harland. "Evaluating the Dynamic Behaviour of Python Applications". In: *ACSC*. 2009, pp. 17–22.

[36] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. "ReIm & ReImInfer: Checking and Inference of Reference Immutability and Method Purity". In: *OOPSLA*. Oct. 2012, pp. 879–896.

[37] Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. "Inference and Checking of Object Ownership". In: *ECOOP*. June 2012, pp. 181–206.

[38] Wei Huang and Ana Milanova. "Inferring AJ types for Concurrent Libraries". In: *FOOL*. Oct. 2012, pp. 82–88.

[39] Wei Huang and Ana Milanova. "ReImInfer: Method Purity Inference for Java". In: *FSE*. Nov. 2012, p. 38.

[40] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM TOPLAS* 23.3 (2001), pp. 396–450.

[41] Marc Fisher II, Jan Wloka, Frank Tip, Barbara G. Ryder, and Alexander Luchansky. "An Evaluation of Change-Based Coverage Criteria". In: *PASTE*. Sept. 2011, pp. 21–28.

[42] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. "Understanding and detecting real-world performance bugs". In: *PLDI*. June 2012, pp. 77–88.

[43] John B Kam and Jeffrey D. Ullman. "Monotone Data Flow Analysis Frameworks". In: *Acta Informatica* 7.3 (1977), pp. 305–317.

[44] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. "Automatic Patch Generation Learned from Human-Written Patches". In: *ICSE*. May 2013, pp. 802–811.

[45] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. "The Soot framework for Java program analysis: a retrospective". In: *CETUS*. Oct. 2011.

[46] Chris Lattner. "LLVM and Clang: Advancing Compiler Technology". In: *FOSDEM*. Feb. 2011.

[47] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Code Generation and Optimization (CGO)*. Mar. 2004, pp. 75–88.

[48] Anatole Le, Ondrej Lhoták, and Laurie J. Hendren. “Using Inter-Procedural Side-Effect Information in JIT Optimizations”. In: *CC*. Apr. 2005, pp. 287–304.

[49] K. Rustan M. Leino, Peter Müller, and Angela Wallenburg. “Flexible Immutability with Frozen Objects”. In: *Verified Software: Theories, Tools, Experiments (VSTTE)*. Oct. 2008, pp. 192–208.

[50] Yin Liu and Ana Milanova. “Static Information Flow Analysis with Handling of Implicit Flows and a Study on Effects of Implicit Flows vs Explicit Flows”. In: *CSMR*. 2010, pp. 146–155.

[51] James R. Low. “Automatic Data Structure Selection: An Example and Overview”. In: *Communications of the ACM* 21.5 (1978), pp. 376–385.

[52] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. 3rd Edition. Addison Wesley, 2005. ISBN: 0321334876.

[53] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. “Parameterized object sensitivity for points-to and side-effect analyses for Java”. In: *ISSTA*. July 2002, pp. 1–11.

[54] Anders Møller. “Static Program Analysis”. Notes. Feb. 2012.

[55] Andrew C. Myers. “JFlow: practical mostly-static information flow control”. In: *POPL*. 1999, pp. 228–241.

[56] David A. Naumann. “Observational purity and encapsulation”. In: *TCS* 376.3 (2007), pp. 205–224.

[57] Stephen Nelson, David J. Pearce, and James Noble. “Profiling Field Initialisation in Java”. In: *RV*. Vol. 7687. LNCS. 2012, pp. 292–307.

[58] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. ISBN: 3540654100.

[59] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. “Practical Pluggable Types for Java”. In: *ISSTA*. July 2008, pp. 201–212.

[60] David J. Pearce. “JPure: A Modular Purity System for Java”. In: *CC*. Mar. 2011, pp. 104–123.

[61] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. “Immutability”. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Vol. 7850. LNCS. 2013, pp. 233–269.

[62] Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. “Inference of Reference Immutability”. In: *ECOOP*. July 2008, pp. 616–641.

[63] Ariel Rabkin and Randy H. Katz. “Static extraction of program configuration options”. In: *ICSE*. May 2011, pp. 131–140.

[64] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. "The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications". In: *ECOOP*. 2011, pp. 52–78.

[65] Gregg Rothermel and Mary Jean Harrold. "Selecting Tests and Identifying Test Coverage Requirements for Modified Software". In: *ISSTA*. 1994, pp. 169–184.

[66] Lukas Rytz, Nada Amin, and Martin Odersky. "A Flow-Insensitive, Modular Effect System for Purity". In: *FTFJP*. July 2013.

[67] Alexandru Salcianu. "Pointer Analysis for Java Programs: Novel Techniques and Applications". PhD thesis. 2006.

[68] Alexandru Salcianu and Martin C. Rinard. "Purity and Side Effect Analysis for Java Programs". In: *VMCAI*. Jan. 2005, pp. 199–215.

[69] Raúl A. Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. "Test-Suite Augmentation for Evolving Software". In: *ASE*. Sept. 2008, pp. 218–227.

[70] Hyunmin Seo, Caitlin Sadowski, Sebastian G. Elbaum, Edward Aftandilian, and Robert W. Bowdidge. "Programmers' Build Errors: A Case Study (at Google)". In: *ICSE*. May 2014, pp. 724–734.

[71] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *USENIX Annual Technical Conference*. 2012, pp. 309–318.

[72] International Organization for Standardization (ISO). "Programming Languages — C++". N3690. May 2013.

[73] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: Fast Detector of Uninitialized Memory use in C++". In: *CGO*. 2015, pp. 46–55.

[74] Arran D. Stewart, Rachel Cardell-Oliver, and Rowan Davies. "Side effect and purity checking in Java: a review". In: *CSSE* (2014).

[75] Bjarne Stroustrup. *The C++ Programming Language*. 3rd Edition. Addison Wesley, 2000. ISBN: 0201700735.

[76] Herb Sutter. *GotW #6a Solution: Const-Correctness, Part 1*. `http://herbsutter.com/2013/05/24/gotw-6a-const-correctness-part-1-3/`. Accessed Dec 2015. May 2013.

[77] LLVM Team. *Checker Developer Manual*. Apr. 2014. URL: `http://clang-analyzer.llvm.org/checker_dev_manual.html`.

[78] LLVM Team. *clang: a C language family frontend for LLVM*. Apr. 2014. URL: `http://clang.llvm.org/`.

[79] LLVM Team. *LLVM Alias Analysis Infrastructure*. Apr. 2014. URL: `http://llvm.org/docs/AliasAnalysis.html`.

[80] LLVM Team. *The LLVM Compiler Infrastructure*. Apr. 2014. URL: `http://llvm.org/`.

[81] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. “Refactoring Using Type Constraints”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33.3 (2011), p. 9.

[82] Matthew S. Tschantz. “Javari: Adding Reference Immutability to Java”. MA thesis. Massachusetts Institute of Technology, 2006.

[83] Matthew S. Tschantz and Michael D. Ernst. “Javari: Adding Reference Immutability to Java”. In: *OOPSLA*. Oct. 2005, pp. 211–230.

[84] Christopher Unkel and Monica S. Lam. “Automatic Inference of Stationary Fields: a Generalization of Java’s Final Fields”. In: *POPL*. Jan. 2008, pp. 183–195.

[85] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. “Soot - a Java Bytecode Optimization Framework”. In: *CASCON*. Nov. 1999, p. 13.

[86] John Whaley and Martin Rinard. “Compositional Pointer and Escape Analysis for Java Programs”. In: *OOPSLA*. Oct. 1999, pp. 187–206.

[87] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. “Automated empirical optimizations of software and the ATLAS project”. In: *Parallel Computing* 27.1–2 (2001), pp. 3–35.

[88] Preston Tunnell Wilson, Justin Pombrio, and Shriram Krishnamurthi. “Can We Crowdsource Language Design?” In: *SPLASH*. Oct. 2017, pp. 1–17.

[89] Jan Wloka, Einar Hoest, and Barbara G. Ryder. “Tool Support for Change-Centric Test Development”. In: *IEEE Software* 27.3 (2010), pp. 66–71.

[90] Jan Wloka, Barbara G. Ryder, and Frank Tip. “JUnitMX — A Change-aware Unit Testing Tool”. In: *ICSE*. May 2009, pp. 567–570.

[91] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. “Precise, Dynamic Information Flow for Database-backed Applications”. In: *PLDI*. 2016, pp. 631–647.

[92] Shin Yoo and Mark Harman. “Regression testing minimization, selection and prioritization: a survey”. In: *STVR* 22.2 (2012), pp. 67–120.

[93] Dmitrijs Zaparanuks and Matthias Hauswirth. “Algorithmic profiling”. In: *PLDI*. June 2012, pp. 67–76.

[94] Qin Zhao, Derek Bruening, and Saman P. Amarasinghe. “Umbra: Efficient and Scalable Memory Shadowing”. In: *CC*. 2010, pp. 22–31.

[95] Qin Zhao, Rodric M. Rabbah, Saman P. Amarasinghe, Larry Rudolph, and Weng-Fai Wong. “How to Do a Million Watchpoints: Efficient Debugging Using Dynamic Instrumentation”. In: *CC*. 2008, pp. 147–162.

Appendices

Appendix A

Clang Mailing List Discussion

Listing A.1: Correspondence with Clang developers.

```
Jon Eyolfson wrote:

Hello Clang Developers,

In my brief time with the clang codebase I've noticed instances of
const_cast and mutable I don't believe are necessary. Are the uses of
const_cast and mutable just temporary to avoid some compile-time type
errors?

> David Blaikie wrote:
>
> Some might be temporary, there are varying levels of 'technical
> debt' that are acceptable/accrued/historical. Some might be
> deliberate/acceptable. It's hard to say generally.

Does const have a different meaning for internal clang code?

> Not especially, though there are perhaps some quirks. One is that if
> an class represents immutable objects then we may not bother using
> 'const' at all, since all instances will always be const, we just
> drop the 'const' from all instances. (see the llvm::Type hierarchy
> for an example of that)

>> Jon Eyolfson wrote:
>>
>> That's a good point, although const wouldn't really hurt either.

>>> David Blaikie wrote:
>>>
>>> Some people (by some people I mean the lead of the project, Chris
```

>>> Lattner) find the const to be just extra textual noise/volume and
>>> prefer not to include it for immutable types.

Is developer time prioritized elsewhere and you're looking for
contributors to help with const-correctness?

> Possibly

The attached example diff demonstrates what I believe is a
const-correct version of CFGBlock::getTerminatorCondition(). I don't
understand the reasoning behind including a non-const version of this
member function that may unintentionally allow bad client code.

> What bad client code do you think this might unintentionally allow?

>> It allows clients to get a Stmt*, which could be used to modify the
>> Stmt (although looking at the Doxygen, Stmt mostly has const member
>> functions). It may be an issue if they use children() and modify
>> something they shouldn't.

>>> That assumes they shouldn't modify them - yes, I suppose certain
>>> modifications would invalidate the CFG, but that's not necessarily
>>> the CFG's responsibility to ensure that. I haven't looked at the
>>> API design enough to know (as a rule, the AST is not mutated, but
>>> there are some narrow cases where it is, I believe). Evidently in
>>> this particular case no client needed to mutate the Stmt at the
>>> moment, but without looking more broadly I wouldn't assume that's
>>> never the case (it might even be valuable to look at what commit
>>> introduced the const/non-const overloads and whether there was a
>>> use-case at some point for the non-const overload).

> I believe this code just provided a const and non-const overload for
> the convenience of users (arguable as to whether that's necessary at
> all, but evidently no one relies on it at the moment if your code
> change compiles). It just happens to be easier to implement the
> overloads by having one overload defer to the other.
>
> Personally I probably would've had the non-const overload defer to
> the const overload (and const_cast the result) as that would be more
> correct than possibly casting away const on an actually const object
> to call a non-const member function from a const member function.

>> What's the reason for the non-const overload at this point? It
>> seems to me that all AST nodes returned by any analysis should
>> always be const. For me, if I wrote "Stmt* S =

>> block->getTerminatorCondition();", I'd rather get a compiler error
>> that it should be "const Stmt* S" then have a const_cast that I
>> personally don't expect.

>>> There is no 'real' const_cast, though - the const_cast is just an
>>> implementation detail used to avoid duplicating the implementation
>>> for const and non-const. Note that CFG already has non-const Stmt
>>> pointers that it could just return without any const_casting.

> Thanks for the reply, Jon

After applying the diff, clang and the default plugins compile
successfully as expected. Granted I don't have any other client code,
so I can't see their uses (or misuses). I would argue that bad client
code would now get a helpful error message instead of possibly really
breaking something in the future. Clients can either fix their code or
be explicit about breaking const and include the const_cast
themselves.

This is my first email to a mailing list, so I apologize in advance
for any misuse.

Jon
